FINAL EVALUATION REPORT
Microsoft, Inc.

Windows NT Workstation and Server
Version 3.5 with U. S. Service Pack 3

NATIONAL
COMPUTER SECURITY CENTER

9800 Savage Road
Fort George G. Meade
Maryland 20755-6000

29 April 1996

Report No. CSC–FER–95/003
Library No. S243,073

This page intentionally left blank

FOREWORD

This publication, the **Final Evaluation Report Microsoft Windows NT** is being issued by the National Computer Security Center under the authority of and in accordance with DoD Directive 5215.1, "Computer Security Evaluation Center." The purpose of this report is to document the results of the formal evaluation of Microsoft Windows NT operating system. The requirements stated in this report are taken from **Department of Defense Trusted Computer System Evaluation Criteria**, dated December 1985.

Approved:

John C. Davis
Director,
National Computer Security Center

29 April 1996

ACKNOWLEDGEMENTS

Team Members

Frank L. Mayer

The Aerospace Corporation
Columbia, Maryland

Maria M. King

Institute for Defense Analysis
Alexandria, Virginia

Gopal Ramanathan

The MITRE Corporation
Bedford, Massachusetts

David K. Black
Cynthia Reese
Jim Reynolds
J. David Thompson

The MITRE Corporation
McLean, Virginia

Steve LaFountain

National Computer Security Center
Fort Meade, Maryland

Acknowledgement is given to Paul Huesken, Ross Lumley, Mark Mercer, and Steve Boczenowski, for their contributions to the evaluation and this document.

TABLE OF CONTENTS

FOREWORD **iii**

ACKNOWLEDGEMENTS **iv**

EXECUTIVE SUMMARY **xv**

1 Introduction **1**
1.1 Evaluation Process Overview . . . 1
1.2 System History . . . 2
1.3 Document Organization . . . 3

2 System Overview **5**
2.1 Executive . . . 5
2.1.1 Hardware Abstraction Layer (HAL) . . . 5
2.1.2 Microkernel . . . 6
2.1.3 Other Executive Services . . . 6
2.2 Protected Servers . . . 8
2.3 Administrator Tools . . . 9
2.4 TCB Interfaces . . . 9

3 Hardware Architecture **11**
3.1 Evaluated Processor Architectures . . . 11
3.1.1 Intel Pentium Processor Architecture . . . 11
3.1.1.1 Processing Units . . . 11
3.1.1.2 Registers . . . 13
3.1.1.3 Execution Architecture . . . 15
3.1.1.4 Memory Segmentation . . . 19
3.1.1.5 Virtual Addressing . . . 21
3.1.1.6 I/O Support . . . 23
3.1.2 DEC Alpha AXP Processor Architecture . . . 23
3.1.2.1 DEC Alpha AXP Processor Hardware . . . 24
3.1.2.2 The Privileged Architecture Library for Windows NT . . . 31
3.1.2.3 I/O Support . . . 34
3.2 Evaluated Machine Architectures . . . 34
3.2.1 Hardware Abstraction Layer . . . 35
3.2.2 Evaluated Machines . . . 36
3.2.2.1 Compaq Proliant 2000 and 4000 . . . 36
3.2.2.2 DECpc AXP/150 . . . 44

4 Software Architecture **49**
4.1 Executive . . . 49
4.1.1 Microkernel . . . 49
4.1.1.1 Dispatcher Objects . . . 50

4.1.1.2 Control Objects . . . 54
4.1.1.3 Interrupt and Exception Handling . . . 55
4.1.2 Object Manager . . . 58
4.1.2.1 Executive Object Type Management Services . . . 58
4.1.2.2 Global Name Space and Related Objects . . . 60
4.1.2.3 Handles and Handle Tables . . . 61
4.1.3 Memory Manager . . . 63
4.1.3.1 Virtual Memory Management . . . 63
4.1.3.2 Physical Memory Management . . . 67
4.1.3.3 Summary of Key Processor Differences . . . 68
4.1.4 Security Reference Monitor . . . 69
4.1.4.1 Tokens . . . 70
4.1.4.2 Access Validation . . . 72
4.1.4.3 Privilege Testing . . . 77
4.1.4.4 Audit Generation . . . 79
4.1.4.5 Logon Session Tracking . . . 81
4.1.5 Process Manager . . . 81
4.1.5.1 Process Definition . . . 81
4.1.5.2 Threads . . . 82
4.1.6 Local Procedure Call Facility . . . 82
4.1.6.1 Port Object . . . 83
4.1.6.2 Client-Server Connection . . . 83
4.1.6.3 Copying a Message to a Port . . . 84
4.1.6.4 Passing a Message in Shared Memory . . . 84
4.1.6.5 Direct Communication . . . 86
4.1.6.6 LPC Impersonation . . . 86
4.1.7 I/O Subsystem . . . 88
4.1.7.1 Single-Layered Device Drivers . . . 90
4.1.7.2 Layered Device Drivers . . . 92
4.1.7.3 File Systems . . . 94
4.1.8 Configuration Manager . . . 102
4.1.8.1 Key Object . . . 102
4.1.8.2 Registry . . . 103
4.1.8.3 Hives . . . 104
4.1.9 Executive Object Services . . . 104
4.1.10 Virtual DOS Machine . . . 105
4.2 Protected Servers . . . 106
4.2.1 Session Manager . . . 107
4.2.2 Security Subsystem . . . 107
4.2.2.1 Local Security Authority . . . 109
4.2.2.2 Security Accounts Manager . . . 112
4.2.3 WinLogon . . . 115
4.2.4 Service Controller . . . 116
4.2.5 Event Logger . . . 117
4.2.6 Win32 . . . 118
4.2.6.1 Win32 Initialization . . . 118
4.2.6.2 Win32 Objects . . . 120
4.2.6.3 Win32 Interface . . . 123

4.2.7 Spooler . . . 126
4.2.7.1 Server . . . 127
4.2.7.2 Printer . . . 127
4.2.7.3 Job . . . 127
4.3 Administrator Tools . . . 128
4.3.1 Backup . . . 128
4.3.2 Chkdsk . . . 128
4.3.3 Control Panel . . . 128
4.3.4 Disk Administrator . . . 129
4.3.5 Event Viewer . . . 129
4.3.6 File Manager . . . 129
4.3.7 Print Manager . . . 129
4.3.8 Program Manager . . . 130
4.3.9 Registry Editor . . . 130
4.3.10 Windows NT Setup . . . 130
4.3.11 User Manager . . . 131
4.3.12 User Profile Editor . . . 131
4.4 Initialization and Shutdown . . . 131
4.4.1 Boot Loader . . . 132
4.4.2 Phase 0: Initialization . . . 132
4.4.3 Phase 1: Software Initialization . . . 133
4.4.4 Phase 2: Software Initialization . . . 134
4.4.5 Shutdown . . . 134
4.4.5.1 User-Mode Shutdown . . . 134
4.4.5.2 Executive Shutdown . . . 134

5 Security Architecture 137
5.1 TCB Protected Resources . . . 137
5.1.1 Users and Groups . . . 137
5.1.1.1 Built-in Accounts . . . 138
5.1.1.2 User Logon and Authentication . . . 138
5.1.2 Subjects . . . 140
5.1.3 Objects . . . 140
5.1.3.1 Object Types . . . 140
5.1.3.2 Object Security Attributes . . . 140
5.1.3.3 Discretionary Access Control List Defaults . . . 142
5.1.3.4 Overview of Object Access Rights . . . 142
5.1.3.5 Object Type-Specific Security Attributes . . . 143
5.2 TCB Security Policies . . . 144
5.2.1 Discretionary Access Control . . . 144
5.2.1.1 DAC Philosophy . . . 144
5.2.1.2 Discretionary Access Control Lists . . . 146
5.2.2 Identification & Authentication . . . 148
5.2.2.1 Identification . . . 149
5.2.2.2 Authentication . . . 151
5.2.3 Privileges . . . 151
5.2.3.1 Privilege Assignment . . . 151
5.2.3.2 Privilege Enforcement and Auditing . . . 152

5.2.3.3 Privilege Attributes . . . 152
5.2.3.4 Privileged Users and Local Groups . . . 152
5.2.4 Object Reuse . . . 153
5.2.4.1 Executive Subsystem Objects . . . 153
5.2.4.2 Protected Server Objects . . . 156
5.2.4.3 Reuse of Hardware . . . 157
5.2.5 Audit Mechanism . . . 158
5.2.5.1 Audit Policy Definition . . . 159
5.2.5.2 Event Logging and Viewing . . . 159
5.2.5.3 Audit Categories . . . 161
5.2.5.4 Audit Records . . . 163
5.2.5.5 Construction of Audit Records . . . 163
5.2.5.6 Audit Loss Prevention . . . 164

6 Assurances **167**
6.1 System Architecture . . . 167
6.1.1 TCB Definition . . . 167
6.1.2 Process and Resource Isolation . . . 167
6.1.3 TCB Integrity . . . 168
6.2 Design Documentation . . . 169
6.3 Test Documentation . . . 169
6.4 Security Testing . . . 169
6.5 System Integrity . . . 170
6.5.1 Intel Pentium Hardware Diagnostics . . . 170
6.5.2 DEC Alpha Hardware Diagnostics . . . 171
6.5.3 Compaq Proliant 2000 and 4000 Hardware Diagnostics . . . 171
6.5.4 DECpc AXP/150 Hardware Diagnostics . . . 173
6.6 Rating Maintenance . . . 174
6.6.1 Personnel Involved in the Microsoft RAMP Process . . . 174
6.6.2 Scope of Product Changes to be Addressed via RAMP . . . 174
6.6.3 RAMP Process Overview . . . 174
6.6.3.1 Windows NT Documentation . . . 174
6.6.3.2 Test Evidence . . . 175
6.6.3.3 Evidence Consistency . . . 175
6.6.4 Configuration Identification and Control . . . 175
6.6.5 Security Analysis . . . 176
6.6.6 Adherence to Interpretations . . . 176
6.6.7 RAMP Audits . . . 176
6.6.8 RAMP Process Failures . . . 176
6.6.9 Hardware Ports . . . 177
6.6.10 Emergency Bug Resolution . . . 177

7 Evaluation as a C2 System **179**
7.1 Discretionary Access Control . . . 179
7.2 Object Reuse . . . 180
7.3 Identification and Authentication . . . 180
7.4 Audit . . . 181
7.5 System Architecture . . . 182

7.6 System Integrity . . . 183
7.7 Security Testing . . . 184
7.8 Security Features User's Guide . . . 184
7.9 Trusted Facility Manual . . . 185
7.10 Test Documentation . . . 185
7.11 Design Documentation . . . 186
7.12 Additional Security Features . . . 186
7.12.1 B2 Trusted Facility Management . . . 186
7.12.2 B2 Trusted Path . . . 187

8 Evaluator's Comments 189
8.1 Robustness of Security When Part of Initial Design . . . 189
8.2 Object Deletion Auditing . . . 189

A Evaluated Hardware Components 191
A.1 Compaq Proliant 2000 and 4000 . . . 191
A.2 DECpc AXP/150 . . . 192

B Evaluated Software Components 193

C Bibliography and References 195

D List of Acronyms 197

This page intentionally left blank

FIGURES

2.1 Windows NT System Overview . . . 6
2.2 Windows NT TCB Interfaces . . . 10

3.1 Intel x86 Functional Units . . . 12
3.2 Intel x86 Flags Register. . . . 15
3.3 Intel x86 Segment Translation . . . 20
3.4 Intel x86 Paging Address Translation . . . 22
3.5 Block Diagram of the Alpha 21064-AA . . . 24
3.6 Compaq Proliant Main System Board . . . 37
3.7 Memory Bit Routing for AECC 4-Bit Protection . . . 38
3.8 Compaq Proliant Processor Module . . . 41
3.9 DECpc AXP/150 System Block Diagram . . . 45
3.10 SYSCTL and HAE Registers . . . 46

4.1 Thread States . . . 52
4.2 Masking Interrupts . . . 57
4.3 Windows NT Virtual Address Translation (4 KB Page Size) . . . 65
4.4 Access Mask . . . 74
4.5 Access Determination Example . . . 78
4.6 Audit Flow . . . 79
4.7 Client-Server Connection . . . 85
4.8 Communication using Sections . . . 87
4.9 The I/O Subsystem . . . 88
4.10 Device Object . . . 89
4.11 Driver Object . . . 90
4.12 Opening an Windows NT File . . . 92
4.13 Device and Driver Objects . . . 93
4.14 Mass Storage Devices . . . 94
4.15 Master File Table . . . 97
4.16 Small File and Directory MFT Entry . . . 98
4.17 Large File MFT Entry . . . 99
4.18 File Objects . . . 100
4.19 Major Security Components of Windows NT . . . 108
4.20 Win32 Objects and Their Attributes . . . 122
4.21 Win32 Server and Client Communication . . . 124

5.1 Audit Flow . . . 158

This page intentionally left blank

TABLES

3.1 Intel x86 Exceptions and Interrupts . . . 17
3.2 Privileged Instructions. . . . 19
3.3 Translation Buffer Page Table Entry . . . 29
3.4 Processor Status Register . . . 31
3.5 Privileged Windows NT Call PAL Functions . . . 32
3.6 Virtual Address Map . . . 32
3.7 Virtual Address . . . 33
3.8 Page Table Entry Fields . . . 33
3.9 Interrupt Distribution in Multi-Processor Compaq Systems . . . 40
3.10 Compaq Model Features . . . 42
3.11 Alpha AXP Memory Map . . . 44

4.1 Priority Levels . . . 52
4.2 Interrupt Request Levels (IRQLs) . . . 56
4.3 Token Data Structure . . . 71
4.4 ACE Format . . . 74
4.5 Security Descriptor Data Structure . . . 75
4.6 Device Driver Entry Points . . . 91
4.7 System Defined File Attributes . . . 98
4.8 Key Object Structure . . . 102
4.9 Token for Processes Running with the SYSTEM Security Context . . . 107
4.10 User Information . . . 114
4.11 Group Information . . . 114
4.12 Fields of an Event Header . . . 117
4.13 Security Relevant User 32 DLL Functionality . . . 126

5.1 Built-in Local Group Accounts . . . 139
5.2 Windows NT Protected Objects . . . 141
5.3 Fields in the SID Data Structure . . . 149
5.4 Well-Known Hard-coded SIDs in Windows NT . . . 150
5.5 Privileges . . . 154
5.6 Windows NT Privileged Built-in Local Groups . . . 155
5.7 Audit Event Types . . . 160

This page intentionally left blank

EXECUTIVE SUMMARY

The security protection provided by Microsoft Windows NT Workstation and Server Version 3.5 with Service Pack 3 when configured according to the Windows NT Trusted Facility Manual, has been evaluated by the National Security Agency (NSA). The security features of Microsoft Windows NT Workstation and Server were examined against the requirements specified by the Department of Defense Trusted Computer System Evaluation Criteria (TCSEC) dated December 1985 to establish a rating.

The evaluation team has determined that the highest class for which Microsoft Windows NT Workstation and Server Version 3.5 with Service Pack 3 satisfies all the specified requirements of the TCSEC is C2. In addition, the B2 Trusted Path and B2 Trusted Facility Management functional requirements are also satisfied.

A system that has been rated C2 provides a Trusted Computer Base (TCB) that enforces a Discretionary Access Control (DAC) policy to protect information and allow users to share information under their control with other specified users, identification and authentication of users to control access to the system and enforce accountability, the prevention of access to residual information from a previous user's actions, and the auditing of security related events. A system that satisfies the B2 Trusted Path requirement supports a trusted communication path between the TCB and the user for identification and authentication. A system that satisfies the B2 Trusted Facility Management requirement supports the ability to separate operator and administrator functions.

Microsoft Windows NT Workstation and Server Version 3.5 with Service Pack 3 is a preemptive multitasking multiprocessor operating system which runs on both CISC (Intel Pentium) and RISC (DEC Alpha) hardware architectures. The platforms in the evaluated configuration are: the Compaq Proliant 2000 and 4000 and the DECpc AXP/150.

This page intentionally left blank

Chapter 1
Introduction

In April 1993, the NSA conducted an Intensive Preliminary Technical Review (IPTR) of the Microsoft Windows NT operating system, a product of the Microsoft Corporation. This report gives evidence and analysis of the security features and assurances provided by Microsoft Windows NT Workstation and Server Version 3.5 with Service Pack 3. Service Pack 3 consists of bug fixes and enhancements to version 3.5 of the Windows NT operating system. Throughout the remainder of this report, it is implied that all references to the Microsoft Windows NT Workstation and Server Version 3.5 products include Service Pack 3. This report documents the evaluation team's understanding of the product's security design and appraises its functions and integrity against the C Division security requirements of the TCSEC.

Material for this report was gathered by the NSA evaluation team through documentation, interaction with system developers, and testing.

1.1 Evaluation Process Overview

The Department of Defense Computer Security Center was established in January 1981 to encourage the widespread availability of trusted computer systems for use by facilities processing classified or other sensitive information. In August 1985 the name of the organization was changed to the National Computer Security Center. In order to assist in assessing the degree of trust one could place in a given computer system, the DoD Trusted Computer System Evaluation Criteria (TCSEC) was written. The TCSEC establishes specific requirements that a computer system must meet in order to achieve a predefined level of trustworthiness. The TCSEC levels are arranged hierarchically into four major divisions of protection, each with certain security-relevant characteristics. These divisions are in turn subdivided into classes. To determine the division and class at which all requirements are met by a system, the system must be evaluated against the TCSEC by an NSA, Trusted Product and Network Security evaluation team.

The NSA supports the creation of secure computer products in varying stages of development from initial design to those that are commercially available. Preliminary to an evaluation, products must go through the Proposal Review Phase. This phase includes an assessment of the vendor's capability to create a secure system and complete the evaluation process. To support this assessment, a Preliminary Technical Review (PTR) of the system is done by the NSA. This consists of a quick review of the current state of the system by a small, but expert, team and the creation of a short report on the state of the system. If a vendor passes the Proposal Review Phase they will enter a support phase preliminary to evaluation. This support phase has two steps, the Vendor Assistance Phase (VAP) and the Design Analysis Phase (DAP). During VAP, the newly assigned team reviews design specifications and answers technical questions that the vendor may have about the ability of the design to meet the requirements. A product will stay in VAP until the vendor's design, design documentation, and other required evidence for the target TCSEC class are complete and the vendor is well into implementation. At that time, the support moves into DAP.

The primary thrust of DAP is an in-depth examination of a manufacturer's design for either a new trusted product or for security enhancements to an existing product. DAP is based on design documentation and information supplied by the industry source, it involves little "hands on" use of the system, but during this phase the vendor should virtually complete implementation of the product. DAP results in the production of an Initial Product Assessment Report (IPAR) by the NSA assessment team. The IPAR documents the team's understanding of the system based on the information presented by the vendor. Because the IPAR contains proprietary information and represents only a preliminary analysis by the NSA, distribution is restricted to the vendor and the NSA.

Products that have completed the support phase with the successful creation of the IPAR, enter formal evaluation. Products entering formal evaluation must be complete security systems. In addition, the release being evaluated must not undergo any additional development. The formal evaluation is an analysis of the hardware and software components of a system, all system documentation, and a mapping of the security features and assurances to the TCSEC. The analysis performed during the formal evaluation requires "hands on" testing (i.e., functional testing and, if applicable, penetration testing). The formal evaluation results in the production of a final report and an Evaluated Products List entry. The final report is a summary of the evaluation and includes the EPL rating which indicates the final class at which the product satisfies all TCSEC requirements in terms of both features and assurances. The final report and EPL entry are made public.

After completion of the Formal evaluation phase, products rated at B1 and below enter the rating maintenance phase (RAMP). The rating maintenance phase provides a mechanism to entend the previous rating to a new version of an evaluated computer system product. As enhancements are made to the computer product the ratings maintenance phase ensures that the level of trust is not degraded.

Rating Maintenance is accomplished by using qualified vendor personnel to manage the change process of the rated product during the maintenance cycle. These qualified vendor personnel must have strong technical knowledge of computer security and of their computer product. These trained personnel will oversee the vendor's computer product modification process. They will demonstrate to the Trusted Product and Network Security Evaluation Division that any modification or enhancements applied to the product preserve the security mechanisms and maintain the assurances required by the TCSEC for the rating previously awarded to the evaluated product.

1.2 System History

The Microsoft Corporation was founded in 1975 and has developed many commercial system-like products such as: Microsoft Disk Operating System (MS-DOS), MS Windows, Lan Manager, and various compilers. They have also developed applications such as Excel, Word, Works, and Powerpoint. Other Microsoft products include the MS Mouse, MS Sound Board, and numerous publications.

In the 1980's Microsoft and IBM developed the OS/2 operating system. This experience led Microsoft to begin development of their own operating system in 1988: Windows new technology (Windows NT). Two of Windows NT's major design goals were to be portable and to exploit both the CISC and RISC hardware advances. The design team of Windows NT included leading experienced persons in VMS, UNIX, and OS/2. Part of the design team's analysis included an examination of the Mach operating system. Another Windows NT design goal was to meet the TCSEC C2 set of requirements.

Coding of Windows NT began in 1989. The first and second Beta versions were released in October and March of 1992, respectively. The first version of Windows NT (Microsoft Windows NT version 3.1) was released in August of 1993. The first and second Beta versions of the second version of Windows NT (version 3.5) were released in March and May of 1994, respectively. The final version of Windows NT (version 3.5) was released in September of 1994. However, since the final release of the operating system several service packages have been released which include bug fixes and enhancements. The evaluated configuration includes Service Pack 3 (which includes Service Packs 1 and 2).

The major difference between the Windows NT Workstation and the Windows NT Server products is that the Windows NT Server offers additional tools to assist in the administration when in a network environment. Some other differences between these two products are: the Windows NT Workstation supports only a single or dual processor while the Windows NT Server supports up to four processors and the Windows NT Server provides fault tolerance features. The evaluated configuration does not include a network environment—both products are considered stand-alone workstations. Therefore, the differences between the Windows NT Workstation and the Windows NT Server that are security relevant in the evaluated configuration are minimal. Consequently, these two products will collectively be referred to as Windows NT, and Windows NT Workstation and the Windows NT Server will each be used to refer to them individually when distinctions are relevant. Additionally, all references to Windows NT include Service Pack 3.

Windows NT falls into Microsoft's Windows family of products which began with Windows 3.1 and Windows for Workgroups 3.1. Windows NT is backward compatible and can run Windows and MS-DOS applications.

There are a few system components which are not included in the evaluated configuration. The Portable Operating System Interface (POSIX) component is eliminated because its UNIX-based security features have not been integrated well with the security features of the Windows NT operating system. The Operating System 2 (OS/2) component is eliminated because it is based on the OS/2 operating system which was not designed to meet security concerns. The networking components of Windows NT are eliminated in this configuration, however, they will be addressed in a follow-on evaluation.

1.3 Document Organization

This report consists of eight sections, four appendices, a bibliography, and a list of acronyms. Section 1 is an introduction. Sections 2 through 6 provide an overview of the system, its hardware and software architecture, and a description of the security support (TCB protection mechanisms and assurances). Section 7 provides a mapping between the requirements specified in the TCSEC and the system features that fulfill those requirements. Section 8 contains additional comments from the evaluation team about the system.

Appendices A and B enumerate the hardware and software components of the evaluated configuration.

This page intentionally left blank

Chapter 2

System Overview

Microsoft Windows NT is an operating system designed to meet the C2 requirements. In the evaluated configuration, Windows NT supports MS-DOS, Microsoft Windows 3.1, and Microsoft 32-bit Windows application execution environments. The evaluated configuration of Windows NT runs on the Intel Pentium (uniprocessor and multiprocessor) and the DEC Alpha based computer systems. The evaluated configuration includes both the Windows NT Workstation and the Windows NT Server products, henceforth collectively called Windows NT.

The Windows NT Trusted Computing Base (TCB) includes the following system components:

- The *executive* which runs in the processor privileged state, called *kernel-mode*
- The *protected servers*, which run in the processor unprivileged state, called *user-mode*
- The *administrator tools*, which also run in user-mode

Figure 2.1 illustrates the Windows NT high-level architecture.[1] Within the executive component, the parts are called *subsystems*.

2.1 Executive

The Windows NT executive is divided into three conceptual layers: the *Hardware Abstraction Layer (HAL)*, the *Microkernel*,[2] and the rest of the executive which includes the following subsystems: I/O (which includes the file systems, I/O Manager, Cache Manager and device drivers), Object Manager, Security Reference Monitor, Process Manager, Executive Object Services, Local Procedure Call (LPC) Facility, Configuration Manager, and Virtual Memory Manager.

2.1.1 Hardware Abstraction Layer (HAL)

HAL provides an abstract view of the underlying machine architecture to the executive, allowing for a machine-independent implementation of much of the Windows NT executive (thereby allowing Windows NT to be easily ported within machines of similar architectures). HAL abstracts machine-specific details for many functions including device I/O, processor initialization, and interrupts.

[1] The layout of the protected servers and administrator tools is not intended to imply layering, as is the case within the executive.

[2] The term "kernel" has several meanings in the context of Windows NT. In much of the Windows NT documentation, kernel means microkernel. However, in some documentation, kernel may refer the entire executive (because the entire executive runs in kernel-mode), or the core set of routines within the Win32 server (because Windows for DOS has a "kernel"). To avoid confusion within this report, we will avoid the use of the term "kernel," with the exception of the term kernel-mode used to refer to processes running in the processor privileged state.

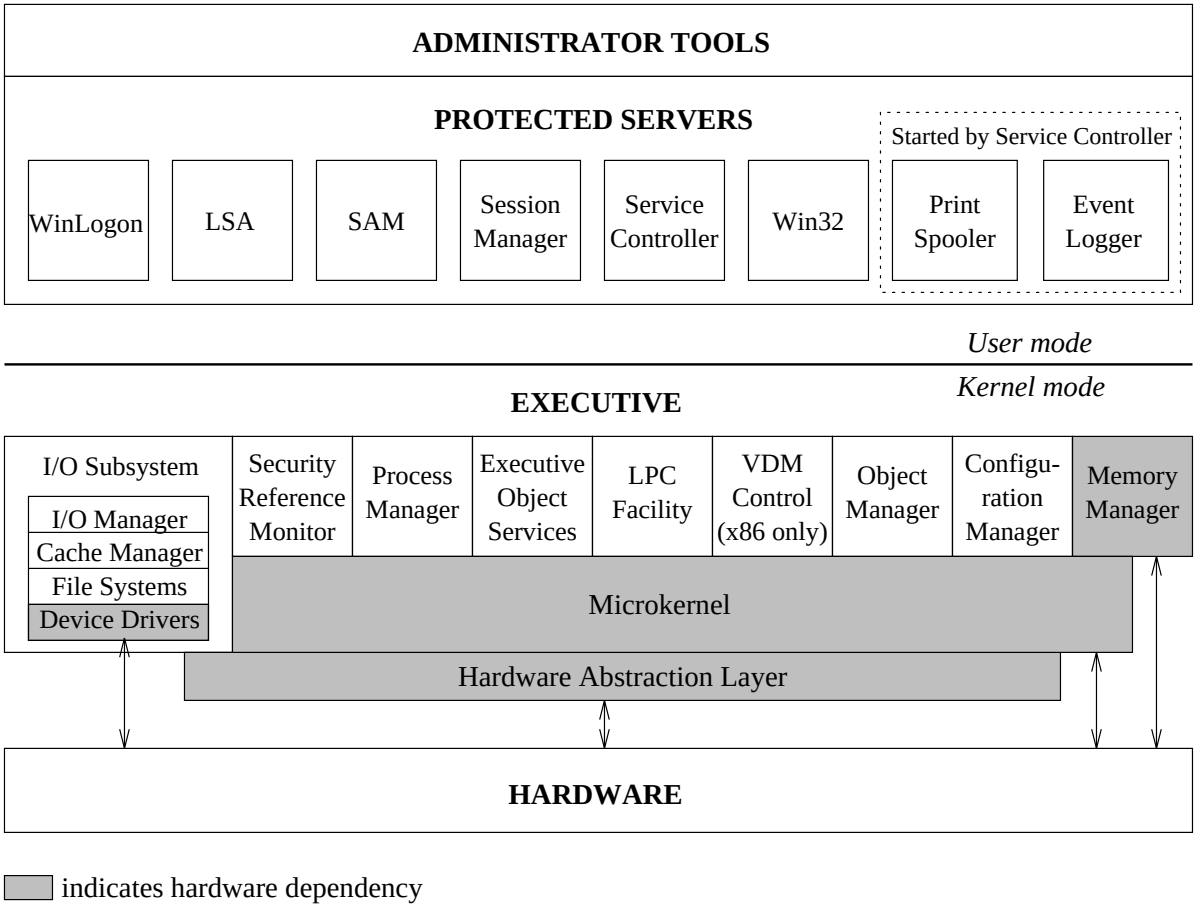


Figure 2.1. Windows NT System Overview

2.1.2 Microkernel

The Microkernel provides a set of primitives to the other executive subsystems. Unlike the rest of the executive, the Microkernel is neither pageable nor preemptable. The Microkernel provides low-level support for execution (e.g., threads, scheduling, context switching), interrupt and exception handling, and synchronization. Some of the Microkernel's exported objects are only used within the executive, but many of the primitive objects it creates are exported to user-mode programs by the Executive Object Services and the Process Manager.

2.1.3 Other Executive Services

The Object Manager is the executive's focal point for creating, accessing, and deleting objects used within the executive and exported outside the executive. While the Object Manager does not implement the semantics of all executive objects, it is used by all executive subsystems as the hub for exporting objects to user-mode programs. As such, it is also responsible for ensuring that access control is enforced. The Object Manager

directly implements three types of executive objects: *type, directory,* and *symbolic link.* The object type "type" provides the ability for executive services to create classes of objects (e.g., directories, files, processes, semaphores) and define their semantics. The object type "type" cannot be exported from the executive.

The Object Manager also implements the notion of *handles* and *handle tables.* In order for a process to access an executive object, it must acquire a handle for that object. Every process has a private handle table (managed by the Object Manager) that contains handles to those objects currently accessed by the process. To acquire a handle, the process must name the object,[3] which resolves to a unique object of a type managed by an executive subsystem. That executive subsystem attempts to create a handle for the requested object by calling the Object Manager. The Object Manager in turn calls the Security Reference Monitor to ensure that the process has access to the object. If access is allowed, the Object Manager creates a handle for the requested object, and puts the handle in the process' handle table. Subsequent usage of the object by the process is via the handle. Among other information, a handle contains the type of access granted (e.g., read, write, delete). A process can only use a handle to perform actions that are allowed by the handle (e.g., a process could not use a read-only handle to write an object—it must first acquire a write access handle). In this manner, the Object Manager is able to ensure that objects are not accessed unless the proper access control decision is made.

The Security Reference Monitor subsystem implements the tests for access control and privilege validation, and is responsible for security audit generation. For executive objects, the Object Manager calls the Security Reference Monitor to validate access before creating a new handle. Other parts of the executive may call the Security Reference Monitor to check for possession of privileges. The Security Reference Monitor also exports security services to user-mode processes. These services are used by the protected servers (e.g., Win32) to perform access validation, privilege possession checks, and audit generation for user-mode objects. The Security Reference Monitor exports a *token* object, which is used to identify a process and its privileges during access and privilege validation, as well as auditing.

The Windows NT executive provides multi-threaded processes. Essentially, a *process* is a private virtual address space, associated physical memory, and a set of accessible objects (i.e., a handle table). A *thread* is a point of execution within a process, and includes all the necessary execution context information (e.g., registers, stack pointers). A process may have zero or more threads (a process with zero threads does not execute). The Process Manager exports processes and threads as executive objects (i.e., they can be "opened" directly via handles if access is granted).

The Virtual Memory Manager subsystem implements a demand-paged memory management subsystem in the Windows NT executive. Each process has a private page table directory which points to some number of page tables (which in turn point to pages of memory). Address translations of virtual addresses use the page table directories and page tables to address real memory. The Virtual Memory Manager subsystem also provides page-level memory protection for read, write, and execute access.[4] The Virtual Memory Manager also exports (via the Object Manager) shared memory *sections* (i.e., portions of memory sharable between two processes) as an executive object type.

The LPC Facility supports communication between processes. The LPC facility exports an executive object type called *ports.*

[3] The name space for executive objects is partially managed by the Object Manager (e.g., for directory objects) and partially by the individual executive subsystems which implement other object types (e.g., file systems implement the portion of the name space containing file system objects).

[4] While the Virtual Memory Manager supports a separate execute access on memory pages, none of the currently supported processor architectures provide a pure execute access on memory pages. Hence, execute access is currently synonymous with read access.

The I/O subsystem provides packet-driven, asynchronous I/O with layered device drivers. The major components of the I/O subsystem include the I/O Manager, the file systems, and various device drivers. Depending on the type of device, a device driver may contain several layers, each layer providing different abstract views of the same physical resource, with each driver layer using the services of a lower-level driver. For example, *file systems* are really a large and sophisticated device driver layered upon several other disk device drivers. The I/O Manager coordinates the interactions between device driver layers by providing an I/O Request Packets (IRP) facility. IRPs facilitate asynchronous communication between I/O subsystem components.

In the evaluated configuration, Windows NT supports five file system types: the File Allocation Table (FAT) file system, the Windows NT File System (NTFS), the CD-ROM File System (CDFS), the Named Pipe File System (NPFS), and the Mailslot File System (MSFS). NTFS is a new file system developed specifically for Windows NT. Only NTFS files are protected by discretionary access control; NTFS is not supported on floppy disks. In the evaluated configuration, FAT file systems are only supported on floppy disks; in the DEC Alpha 21064-AA architecture a FAT partition is required for booting. The FAT filesystem, NTFS, and CDFS use the services of the Cache Manager. The Cache Manager supports memory-mapped I/O and provides a memory-based cache which is used by file system drivers to improve performance for certain I/O-bound programs. The Cache Manager uses the Virtual Memory Manager's services and memory sections to provide a cache that grows and shrinks with need and memory availability. The Cache Manager does not directly export any executive object.

The Configuration Manager implements the configuration registry. The registry is used to store various system configuration information. For example, the Security Accounts Manager (SAM) database, hardware configuration and initialization information, and OS configuration information are all maintained by the Configuration Manager in the registry. The registry is implemented as a tree database, indexed by *keys*. Keys, and the portion of the registry to which they refer, are the executive object type exported by the Configuration Manager.

The Process Manager and the Executive Object Services subsystems provide user-mode interfaces for the exported microkernel objects. The Process Manager provides the user-mode interface for process and thread objects. The Executive Object Services subsystem provides the user-mode interface for event, event pair, mutant (under the name "mutex"), semaphore, timer, I/O completion port, and profile objects.

2.2 Protected Servers

Many of the Windows NT security relevant services are provided by user-mode processes called *protected servers*. The protected servers in the evaluated configuration are: WinLogon, the Print Spooler, the Session Manager, the Local Security Authority (LSA) subsystem, the Security Accounts Manager (SAM) subsystem, the Service Controller, and the Event Logger. All protected servers execute in processes that possess privileges.

WinLogon coordinates user authentication. It is the process that prompts for user logon identifiers and passwords, and ensures that proper authentication occurs before allowing a user to perform further actions. WinLogon calls the Local Security Authority (LSA) subsystem to validate the user identity and password. In addition to user authentication, the LSA is also involved in the collection and storage of audit records. The SAM subsystem provides security administration for user accounts.

The Win32 server implements the user interface to Windows NT. It is a separate process that exports two user-mode objects: a *WindowStation* object and a *Desktop* object. A WindowStation object is the means

by which clients access and manipulate the Win32 server-managed resources including the keyboard, mouse, and display. Desktop objects provide abstract resources like menus, windows, and title bars. All Desktop objects are associated with a WindowStation object (i.e., one must first obtain access to a WindowStation object before being able to obtain access to the associated Desktop object). Win32 protects and audits both of these objects in a manner consistent for executive objects by using the executive Security Reference Monitor's access validation, privilege checking, and audit services.

The Print Spooler is a protected server that routes print jobs to the appropriate printer device. It exports a printer object, a job object and a server object.

The Session Manager starts other system processes. During system initialization, the Session Manager starts the WinLogon process and the Win32 protected server. WinLogon (for interactive users), along with the Service Controller (for service logons), supports user logon by interfacing with the Session Manager to activate a user process after the user has been successfully authenticated. The Session Manager sets the security attributes of this new process to that supplied by the logon process, and returns this new process' handle to the logon process.

The Service Controller is a protected server that manages both drivers and services by starting, stopping, and reporting on the status of each. All drivers that specify automatic loading in the registry are loaded by the Service Controller at boot-time.

2.3 Administrator Tools

Windows NT provides several system administration tools. These tools manage all aspects of the Windows NT system, including tools to manage user accounts, audit configuration, system and device configuration, file system backup, and the audit events reviewer. The set of administrator tools included in the evaluated configuration are: Backup, Chkdsk, Control Panel, Disk Administrator, Event Viewer, File Manager, Print Manager, Program Manager, Registry Editor, Setup, and User Manager.

2.4 TCB Interfaces

Figure 2.2 illustrates how untrusted subjects request services from the Windows NT TCB. There are three ways to request TCB services (three types of TCB interfaces):

- Unprivileged hardware instructions
- Windows NT executive system services
- Interprocess Communication (IPC).

An untrusted subject can execute any unprivileged *hardware instructions*, such as arithmetic operations.

An untrusted subject may execute *system services*, which are calls to the Executive. In the $x86$ machines, the execution of a system service results in the execution of an *INT 2E* instruction which causes the transition from user-mode to kernel-mode. In the DECpc AXP/150 machines it is a *callsys* PAL function. In both architectures, the system service call parameters are captured and probed. The capture involves copying the arguments from user mode to kernel mode. The probe involves ensuring that the parameters really came

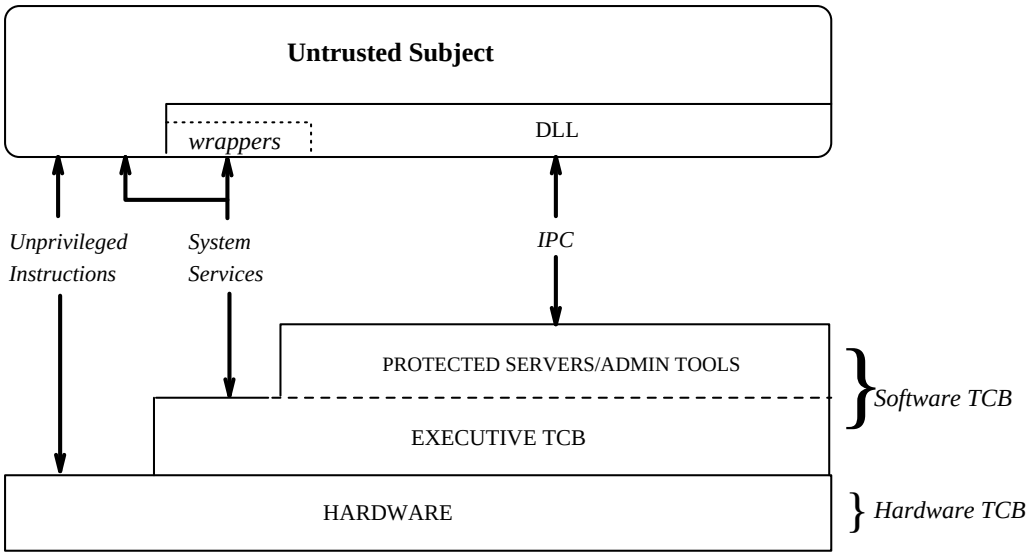


Figure 2.2. Windows NT TCB Interfaces

from the user's address space and are valid in terms of access requirements (See Section "Interrupt and Exception Handling," on page 55).

The third way of requesting TCB services is by calling a TCB protected server (a TCB user-mode process) via inter-process communication. This type of service uses one or more of the Executive's inter-process communication mechanisms between an untrusted client process and a TCB server process.

It should be noted that the advertised programming interface to the Windows NT system is via a large set of Dynamic Link Libraries (DLL). These DLLs, when linked into an untrusted process, are themselves completely untrusted by the TCB. In all cases, the functions implemented by a DLL will result in one or more of the three types of TCB service requests listed above.

Chapter 3
Hardware Architecture

This chapter presents an overview of the hardware architectures included in the evaluated configuration of Microsoft Windows NT Workstation and Server Version 3.5. The processor architectures are described first, followed by descriptions of the machine architectures.

3.1 Evaluated Processor Architectures

The evaluated configuration of Windows NT includes two different processor architectures. The first is the Intel Pentium microprocessor, referred to throughout this report as the Intel $x86$. The second is the Digital Alpha AXP processor line; specifically the DECchip 21064-AA processor.

3.1.1 Intel Pentium Processor Architecture

The Pentium microprocessor, introduced in 1993, is the latest enhancement of the $x86$ family and provides performance improvements over the i486 while maintaining 100% backward compatibility with the entire $x86$ family. Designed using a superscalar RISC architecture, it has two five-stage execution units and can process up to two instructions in a single clock cycle. Separate 8 Kilobyte (KB) code and data write-back caches reduce cache conflicts and increase processor performance over its predecessor. It has internal code and data paths of 256 and 64 bits, respectively. Processor performance is improved by a pair of instruction pipelines, one fetching instructions linearly and one controlled by a branch prediction mechanism.

The $x86$ processor protection mechanism provides four privilege levels, numbered 0 to 3 (see Section "Execution Architecture," on page 15). Privilege level 0 is the most privileged and privilege level 3 is the least privileged. NT uses two of the four privilege levels when running on an x86 system. Kernel mode is x86 privilege level 0 and user mode is x86 privilege level 3.

3.1.1.1 Processing Units

The $x86$ processor has the following processing units (see Figure 3.1):

Bus Interface Unit (BIU). The Bus Interface Unit (BIU) interfaces the $x86$ to the external processor bus. Requests by other processor components for access to the bus (e.g., instruction prefetch, data prefetch, memory reads, and cache fills) are prioritized and executed by the BIU.

Cache Unit. The Pentium has separate internal 8 KB data and 8 KB instruction caches that reduce the need to access external memory. It also includes cache coherency interface logic for external second-level caches.

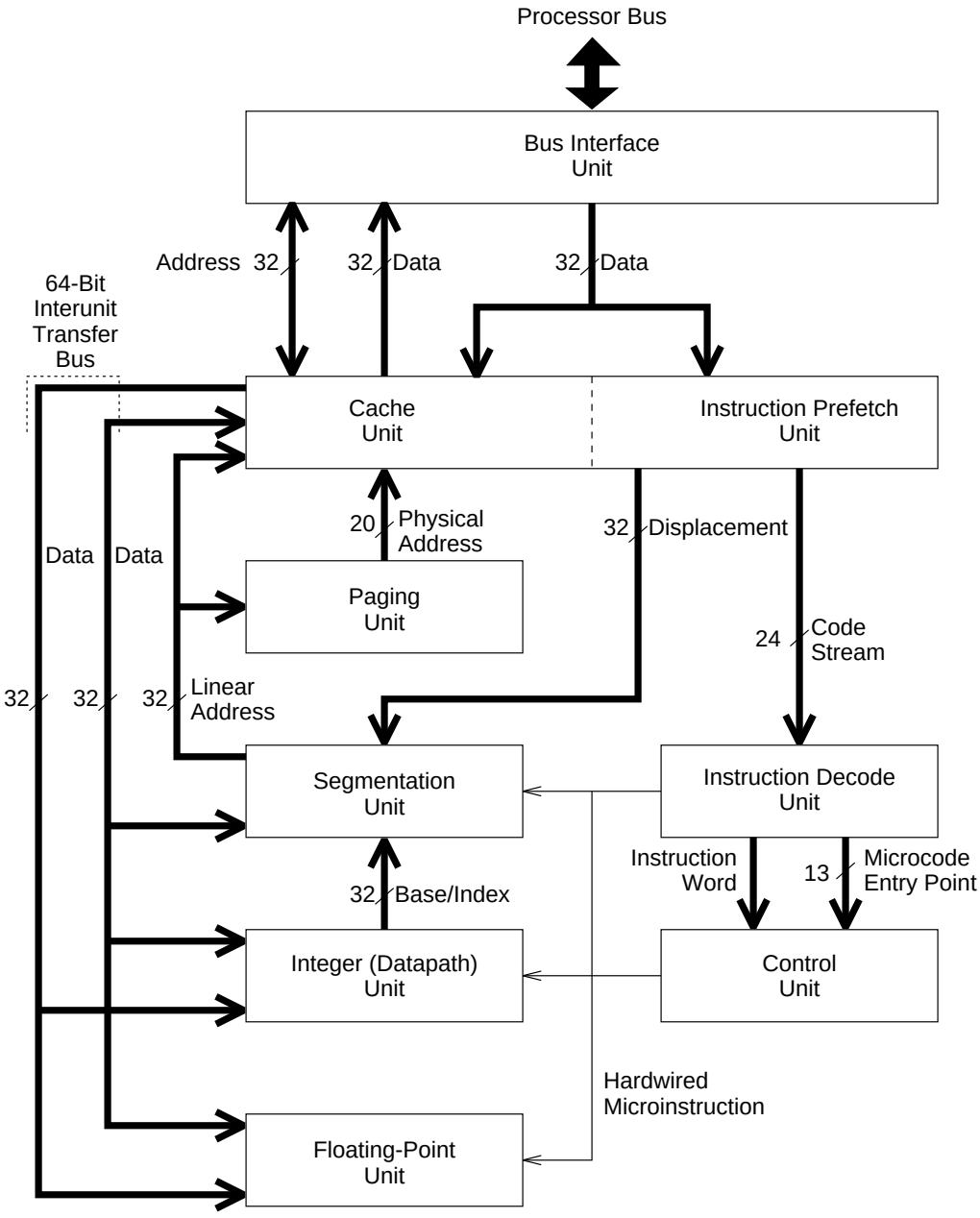


Figure 3.1. Intel x86 Functional Units

Instruction Prefetch Unit. Reads bytes from the instruction stream, via the BIU, during what would otherwise be idle bus cycles. This unit is closely coupled with the cache unit. Prefetched data is read into the memory cache and the prefetch units simultaneously. The prefetch unit is flushed whenever a control transfer, such as a jump, interrupt, or task switch is performed.

Instruction Decode Unit. Reads instructions from the prefetch unit and translates the machine code bytes into control signals for the other processing units.

Control Unit. Executes decoded instructions. It controls the integer, floating point, and segmentation units. The control unit of the *x*86 contains the processor's microcode.

Integer Unit. Contains the processor's eight 32-bit general registers and the arithmetic and logic unit. Two 32-bit data buses connect the integer unit with the memory cache and the floating point unit, and are used to transfer 64-bit operands. A separate 32-bit data bus connects the integer unit with the segmentation unit to transfer data for address generation.

Segmentation and Paging Units. These units cooperate to implement the memory management functions. The segmentation unit translates logical (segmented) addresses into linear (unsegmented) addresses. The paging unit implements a virtual memory scheme which translates linear addresses into physical (hardware) addresses (see Section "Virtual Addressing," on page 21).

Floating Point Unit. This unit is an integrated floating-point computation unit.

3.1.1.2 Registers

The registers of the *x*86 are organized into the following groups: General Registers, Instruction Pointer Register, Segment Registers, Memory Management Registers, Control Registers, Flags Register, Debug Registers, Floating Point Registers, and Test Registers.

Register names beginning with 'E' are extended (32-bit) versions of registers of the same name without the 'E' in earlier Intel processors in the x86 family. These 16-bit non-extended registers are still available in the Pentium, but only in real-mode.

General Registers. There are eight 32-bit general registers named: EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. They can be used as operands for logical and arithmetic operations and for address calculations (except ESP, which cannot be used as an index in an address calculation).

Some of these general registers also have specific uses with certain instructions. The EBP is optimized for use as a book mark into the return stack. ESI and EDI are used implicitly by string operands as byte source and destination indices or pointers. ESP is the default index into the stack segment (pointed to by SS) for all stack functions. The general registers can be changed from user mode.

Instruction Pointer Register. The instruction pointer (EIP) is a 32-bit register that holds the offset into the current code segment of the next instruction to be executed.

Segment Selector Registers. There are six 16-bit segment selector registers used to identify the currently addressable memory segments. SS, DS, ES, FS, and GS can identify data segments. CS identifies the code segment from which the processor will fetch instructions. The segment selector registers can be changed from user mode.

In x86 protected mode, which NT uses exclusively, each segment register has an extension called a segment descriptor register, which is outside of the user-mode programming model. A segment selector specifies an index into a local or global segment descriptor table, from which the segment descriptor is loaded with the segment base address, length, and other attributes. This saves the processor from having to reference the table on each use of the segment selector in evaluating addresses.

User-mode operations can cause a segment descriptor to be changed by changing the value of the corresponding segment selector, but the executive still maintains control of the values to which it can be changed. The segmentation process is described more fully in Section "Memory Segmentation," on page 19.

Memory Management Registers. The Global Descriptor Table Register (GDTR) and the Local Descriptor Table Register (LDTR) point to the active Global and Local descriptor tables used for resolving segmented memory addresses. The Interrupt Descriptor Table Register (IDTR) points to the table of processor interrupt vectors, from which interrupt routine pointers are selected by interrupt number. The task register (TR) points to the Task State Segment (TSS), where the processor stores and loads state information during a task switch. These registers cannot be modified while in user mode.

Control Registers. CR0 contains general processor configuration flags (enables or disables: cache, paging, alignment checking, the write protection of user pages, segment level protection, the floating point unit, numeric instruction emulation, and cache write-through). Flags in CR0 also indicate the state of interrupted floating point unit instructions after a task switch. CR1 is reserved for future use by Intel. CR2 holds the 32-bit linear address that caused the last page fault. CR3 (also called the Page Directory Base Register) points to the first-level page directory table for the current task (see Section "Paging," on page 21). CR4 supports extensions to the virtual-8086 and debugging modes, specifies the page size (4 KB or 4 Megabyte (MB)), and enables or disables user access to the time stamp counter, the virtual interrupt flag in kernel mode, and the machine check exceptions. These registers cannot be modified while in user mode.

The floating point unit has its own set of control registers, including the *status word*, the *control word*, and the *tag word*. The status word includes the zero divide fault, overflow, and stack fault indicators. It also contains a pointer to the top of the floating point register stack. The control word records the current precision setting, round-off setting, and overflow interrupt mask. The tag register is used to control referencing to the floating point registers.

Flags Register. The EFLAGS register is 32-bits in length, is accessible only while in kernel mode and contains flags that control such features as the I/O privilege level, maskable interrupts, and debugging (see Figure 3.2).

Debug Registers. These are 32-bit registers used for implementing the debugging capabilities of the *x*86, such as the setting of breakpoints without modifying code segments. DR0, DR1, DR2, and DR3 are Debug Address Registers used to store the addresses of breakpoints. DR4 and DR5 are reserved by Intel for future use. DR6 is the Debug Status Register, which is set by the processor when an enabled debug exception is detected. This allows the debugger to determine which debug conditions have occurred. DR7 is the Debug Control Register, which is used to specify which actions cause breaks and to enable the Debug Address Registers locally or globally (when paging is used). They may be accessed only while in kernel mode.

Floating Point Registers. The *x*86 floating point unit provides eight 80-bit data registers for floating point calculations. These registers are accessible while in user mode.

31 30 29 28 27 26 25 24 23 22 21 20 19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved by Intel	AC	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

AC: The Alignment Check bit used to indicate that all instruction operand 32-bit memory addresses will be aligned to minimize read operations.
VM: This bit is set when the processor is in Virtual 8086 mode.
RF: The resume flag is used with debugging register breakpoints. These registers are not available to programs except those in privilege level zero.
NT: The nested task bit is set when the processor is in kernel mode and multitasking is allowed (see Section "I/O Support," on page 23).
IOPL: The I/O Privilege Level (IOPL) is used in kernel mode and specifies the privilege level required to perform I/O operations (see Section "I/O Support," on page 23).
OF: The overflow flag is set if an operation resulted in a signed overflow.
DF: The direction flag controls the direction strings are processed.
IF: The INTR enable flag allows external interrupts signaled on the INTR processor pin to be processed. When IF is reset, external interrupts are not recognized.
TF: The trap enable flag is used for single-stepping through code with debug registers.
SF: The sign flag is a copy of the sign bit of a result.
ZF: The zero flag is set if all the bits of a result are zero.
AF: Auxiliary carry is set on a carry out of bit 3 during Binary Coded Decimal (BCD) operations.
PF: The parity flag is set if the low-order eight bits of an operation contain an even number of bits set.
CF: The carry out of the most significant bit of certain arithmetic and logical operations.

Note: The underlined flags can be modified while in user mode. All others may be modified only while in kernel mode.

Figure 3.2. Intel x86 Flags Register.

3.1.1.3 Execution Architecture

The *x*86 supports three addressing modes: Real mode, Protected mode, and Virtual 8086 mode. The processor always starts operating in Real mode.

In Real mode, every task has full and unrestricted access to the lower 1 megabyte of address space. The only way to leave Real mode is to switch to Protected mode (entered when a move instruction sets the protection enable (PE) bit in CR0). The processor can be returned to Real mode with either a RESET signal or by clearing the PE bit in CR0. Windows NT uses Real mode only for initialization. It switches to Protected mode before user processes are started and never returns to Real mode.

All *x*86 features and instructions are available in Protected mode, including virtual memory, memory segmentation, and memory protection. All four hierarchical privilege levels are available (although only two are used in Windows NT), as are memory access mediation, and task separation mechanisms. Except during system initialization, Windows NT operates in Protected Mode.

Virtual 8086 (V86) mode provides the ability to run multiple Real mode processes under Protected mode. Windows NT uses Virtual 8086 mode to encapsulate MS-DOS-compatible processes in Windows NT processes. Virtual 8086 mode gives a process the illusion it is running in Real mode, while preventing it from tampering with the protection mechanisms isolating it from other Protected-mode processes. The *x*86 supports four hierarchical privilege levels, numbered 0 through 3, in decreasing order of privilege. One privilege

level (at a time) is associated with every task and memory descriptor. Windows NT uses only two of these levels. Level 0 is kernel mode and level 3 is user mode.

Privilege levels provide hardware supported memory protection. Using the privilege levels, Windows NT can provide separation between user processes and the operating system and allow the operating system to perform actions that are not permitted by users. The key feature of the protection mechanism is the *Selector*. The selector is used by tasks to access system objects such as memory segments, tables supporting the protection mechanism, special segments that store the processor state, and access control objects called *gates*. Information about the object referenced by the selector is contained in *descriptors*. Descriptors are grouped in descriptor tables. The descriptor stores the privilege level of the object it references. The privilege level of the selector in the CS register defines the Current Privilege Level (CPL) of the currently executing process.

The following privilege rules are enforced by the $x86$ when operating in Protected Mode.

- Data can be accessed by processes with privilege level (CPL) equal to or less than that of the data segment (DPL) (i.e., with equal or greater privilege).
- Code can be executed (i.e., a segment descriptor may be loaded into the CS register) by processes with privilege level (CPL) equal to or greater than that of the code segment (DPL) (i.e., with equal or lesser privilege).

Any attempt to reference code or data by a task with insufficient privilege will cause a trap to occur before any memory is referenced or modified.

Each privilege level has its own stack, determined by the setting of the current privilege level (CPL) of the executing task and the TSS register. The TSS contains stack memory segment information for privilege levels 0–2. There is no stack information for privilege level 3 because it is the lowest level and cannot be transferred into from a lower level. Since Windows NT uses privilege levels 0 and 3 only, a single TSS is maintained for user-mode transitions into kernel mode.

3.1.1.3.1 Exception Handling and State Transitions Task execution can be suspended by an interrupt or a trap. In Windows NT, interrupts can be generated by external hardware signals or by internal software instructions. Traps, on the other hand, arise when an instruction cannot be completed normally (e.g., because of a page fault during execution). The $x86$ provides the capability to specify up to 256 interrupt entry points. The complete list of $x86$ interrupts is in Table 3.1. Windows NT uses the Call to Interrupt Procedure instruction to vector 46 (INT 46) to implement calls to the executive. In Protected Mode this instruction (INT 46) generates a call to the executive through a task gate. It is the means by which NT user-mode process' request services of the executive.

Interrupts and traps force the transfer of control from the executing task to the interrupt handler. The interrupt handler performs any actions necessary to satisfy the cause of the interrupt. The task switch causes the old task state to be pushed onto the stack, from which it will be restored when the interrupt handler executes an interrupt-return (IRET) instruction.

Gate descriptors provide the ability to transfer control between code segments, and to change a process' privilege level. Gate descriptors are special types of memory descriptors that contain pointers to destination segment selectors instead of the addressing data contained in a normal segment descriptor. There are four types of gates: call, trap, interrupt, and task. Call gates can only be used to transfer control to equally or

Event	Type	Description
0: Divide Error	Exc	Occurs on an integer divide by zero.
1: Debug Exception	Exc	Occurs when debug condition are satisfied.
2: NMI Interrupt	Int	Reserved for the hardware non-maskable interrupt condition (memory fetch errors).
3: Breakpoint	Exc	Used by debuggers to trap breakpoints.
4: Overflow	Exc	Occurs when the processor executes an INTO instruction with the OF flag set.
5: Bounds Check	Exc	Generated when the processor, while executing a BOUND instruction (which compares an array index with an upper bound), finds that the operand exceeds the specified limits.
6: Invalid Opcode	Exc	Generated when an invalid opcode or operand is detected by the execution unit.
7: Coprocessor Not Available	Exc	Generated when an FPU instruction is attempted and the FPU has been locked (the TS bit is set in CR0).
8: Double Fault	Exc	Generated when certain faults occur while processing a previous fault.
9: Coprocessor Segment Overrun	Exc	Signaled when a floating point instruction causes a memory access that runs beyond the end of a segment.
10: Invalid Task State Segment	Exc	Generated on a task switch to a segment with an invalid TSS.
11: Segment Not Present	Exc	Generated when the processor detects that the present bit of a descriptor is clear. Used by some operating systems (not NT) to implement virtual memory via the segmentation mechanism.
12: Stack Fault	Exc	Generated by a limit violation in any operation which refers to the SS register or attempts to load the SS register with a descriptor that is marked segment-not-present but is otherwise valid.
13: General Protection	Exc	Occurs for all other protection violations.
14: Page Fault	Exc	Occurs when paging is enabled (the PG bit in CR0 register is set) and the processor detects that a page table or page is not present in physical memory or the task does not have sufficient privilege to access the indicated page.
15:	Exc	Intel Reserved.
16: Floating-point error	Exc	Signals an error generated by a floating-point instruction.
17: Alignment Check	Exc	Generated by an access to an unaligned operand.
18–31:	Exc	Intel Reserved.
32–42:	Exc	Unused.
43: KiSetHighWaitLowThread service	Exc	Used with Int 44 to implement event-pairs.
44: KiSetLowWaitHighThread service	Exc	Used with Int 43 to implement event-pairs.
45: Debugger Calls	Exc	Used by the debugger.
46: NT System Service Calls	Exc	Used by NT to trap user-mode system service calls.
47: APIC	Int	Asynchronous Programmable Interrupt Controller.
48–255:	Exc	Not dedicated.

Table 3.1. Intel x86 Exceptions and Interrupts

more privileged segments (i.e., to segments with DPL $\leq$ CPL). Loading a destination segment selector into the CS register causes the Descriptor Priority Level (DPL) of the segment descriptor to be loaded into the CPL of the CS register, thereby changing the privilege level of the process. A stack switch will also occur if the above process results in a change in the CPL.

An interrupt gate is associated with each of the 256 interrupt entry points. Descriptors for these gates are held in the IDT (see Section "Memory Segmentation," on page 19). The gates in the IDT may be interrupt, trap, or task gates. Interrupt and trap gates are used to handle exceptions. Each contain selectors, which point to the target code segment, and an offset into that segment. Task gates are used to execute task switches and contain a pointer to a task's segment.

A stack switch is necessary for gated transfers to more privileged levels in order to protect the system from a potential overflow of the user's stack, and to protect the stack values from modification by another of the user's threads while the privileged operation is underway. The contents of the processor registers and a number of parameters specified in the gate descriptor are saved on the stack, and will be restored on return from the privileged process. The original privilege level will be restored when privileged processing is complete.

Interrupts are either maskable or non-maskable. Maskable interrupts are received on the interrupt request line (INTR) driven by external logic. Non-Maskable Interrupts (NMIs) are received on the NMI line and are used to signal catastrophic events, such as a memory failure. Windows NT halts the processor on the detection of an NMI, since the integrity of the machine is in doubt and any other action may further damage the system or the user's data. NMIs are only disabled during the processing of another NMI.

3.1.1.3.2 Privileged Instructions The *x*86 instruction set includes a number of instructions which affect the protected state of the processor and can be executed only in kernel mode. A general protection exception is generated if the execution of one of these instructions is attempted when the processor is not in kernel mode (see Section "Execution Architecture," on page 15). The complete list of *x*86 privileged instructions is in Table 3.2 The protection of these instructions prevents user-mode processes from: denying service to other users (HLT), impacting the machine's performance (INVD, INVLPG, WBINVD), or subverting the memory protection mechanisms (LGDT, LIDT, LLDT, LMSW, LTR, CLTS, and the MOVs to CRn, DRn, and TRn).

3.1.1.3.3 Cache The *x*86 contains an on-chip cache for storing 8 KB of instructions and data. The Pentium contains separate 8 KB caches for instructions and data. The cache improves processor performance by satisfying many read requests more quickly from on-chip cache than it could from a memory bus cycle. The internal cache is transparent to program operation.

Since internal cache lines[1] are tagged with physical addresses, changes to the paging tables (as a result of process swapping or memory allocation) can alter the relationship between linear and physical addresses and render the cache mapping invalid. In order to prevent the use of invalid cache entries, the operating system must force the caches to be marked as invalid after any page assignment change. This is accomplished with the WBINVD and INVD processor instructions.

[1] Cache lines refer to the smallest amount of memory associated with a cache address tag. The *x*86 first-level cache uses 32-byte lines. Therefore, the lower five bits of an address is not compared against the line's tag, but used to select the correct byte from the line whose tag does match the rest of the address bits.

Instruction	Description
CLTS	Clear Task-Switched Flag
HLT	Halt the processor
INVD	Invalidate cache
INVLPG	Invalidate Translation Lookaside Buffer page entry
LGDT	Load Global Descriptor Table
LIDT	Load Interrupt Descriptor Table
LLDT	Load Local Descriptor Table
LMSW	Load Machine Status word
LTR	Load Task register
MOV to/from CRn	Move to/from Control Register n
MOV to/from DRn	Move to/from Debug Register n
WBINVD	Write back and invalidate cache

Table 3.2. Privileged Instructions.

WBINVD causes all modified data cache lines to be written back to memory and all lines in both caches to be marked as invalid. It also drives the Writeback and Flush processor signals to the memory subsystem to signal external caches to write back their modified entries and invalidate their cache lines. The INVD instruction is similar to WBINVD except that it does not write back modified lines and does not drive the Writeback processor signal.

3.1.1.4 Memory Segmentation

The *x*86 processor does Protected mode segmented address translation regardless of whether or not memory paging is enabled. Each address space is referenced by a logical address consisting of a segment selector and an address offset. The segmentation mechanism uses the segment selector as an index into a segment descriptor table to find the segment base address in the linear address space. It then adds the offset in the instruction to produce a linear address. For any address translation, the segment selector to be used is contained in a segment register (see Figure 3.3).

The access rights field in the segment descriptor contains flags which define the segment as containing either code or data and indicate whether it is read-only, read/write, execute, execute/read, etc. It also contains the segment's privilege level.

Segment descriptors define the base location, extent, and access rights of the segment in the linear address space of the *x*86. A segment base can point to any location in the 4 Gigabyte (GB) logical address space. There are two types of tables: the Global Descriptor Table (GDT) and the Local Descriptor Table (LDT). These tables are located by the GDTR and LDTR registers, respectively. There is one global table for all processes running on the processor, whereas each process has its own local table. Together, a local and global table define the memory context for every process. A user-mode process may load any descriptor from its LDT or the GDT into a segment descriptor by setting one of its segment selectors to specify the table (LDT or GDT) and the offset to the descriptor within that table.

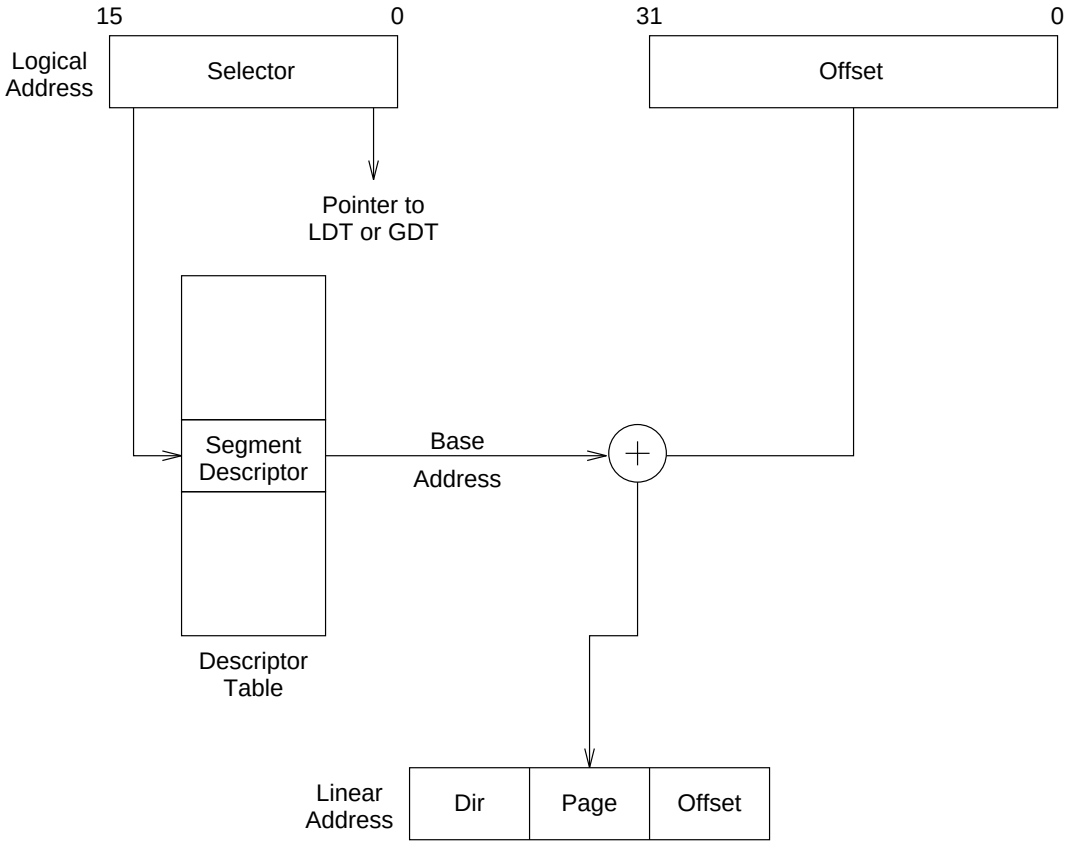


Figure 3.3. Intel x86 Segment Translation

Windows NT uses the $x86$'s pagination features, rather than segmentation, for memory access control. This is primarily because pagination is more widely available on different processor types. $x86$ segmentation cannot be completely disabled, though, because it also implements the privilege level mechanism and is used to implement call gates and traps (and, the $x86$ offers no method for disabling it). Windows NT simplifies the use of segmentation as much as possible by giving each process just two memory segment descriptors, one for data and one for code, each of which describes the process's entire linear address space (i.e., before pagination). The data descriptor can be loaded into any segment register except the CS, and is set to allow read and write access. The code descriptor can be loaded into the CS register, and is set to allow execute access.

Windows NT assigns the DPL field of the user's memory segment descriptors to 3 (user mode) and the DPL of the executive's memory segment descriptors to 0 (kernel mode). When a segment descriptor is loaded into the CS register, the descriptor's DPL becomes the process's CPL. The trap and gate mechanisms use the CPL, the DPL, and the Requested Priority Level (RPL) (from the segment selector) to control the calling of system service routines.

3.1.1.5 Virtual Addressing

The memory accessible on the $x86$ bus is called physical memory. This memory is organized as a sequence of 8-bit bytes with each byte assigned a unique address that can range from zero to 2^{32} (approximately 4 billion).

3.1.1.5.1 Virtual Address Translation The $x86$ provides segmented and paged virtual address translation. Segmentation provides multiple, independent, address spaces. Paging supports a model of a large address space using a small amount of physical memory combined with disk storage. Memory addresses referenced by executing programs are usually logical addresses that are relative to the context of the program itself. The $x86$ uses both segmentation and paging to translate a logical to a physical address with an absolute location. Segmentation hardware is used to translate the logical address into an address for a continuous address space called a linear address. Paging hardware is then used to translate the linear address into a physical address.

3.1.1.5.1.1 Paging Paging gives another level of organization to memory. The $x86$ hardware paging unit uses the 32-bit linear address provided by segmentation along with the page directories and tables to generate a physical address. It breaks the linear address space into fixed-sized 4 KB blocks, called pages. A page may be in memory or on disk. When a linear address is referenced, it is translated into an address for a page in memory, or an exception is generated. An exception gives the operating system an opportunity to read the page from disk and update the page mapping, or to assign a new page to a process' virtual address space. However, the page referenced may be outside the process' virtual address space, in which case an error (i.e., segmentation fault) is returned.

Paging is enabled when the PG bit of CR0 is set by the operating system during system initialization. This causes paged address translation to be performed on all 32-bit linear addresses generated via segmentation. The paging mechanism treats this 32-bit address as having three distinct parts, two 10-bit indexes and a 12-bit offset.

The two 10-bit indexes are used to select entries in the *page directory* and the *page tables*. The 12-bit offset selects a byte address in the referenced page. A page directory entry points to an array of page tables. A page table entry points to physical memory pages. Page directories and tables are arrays of 32-bit entries and occupy a page themselves.

Page directories and tables provide two levels of address translation. The page table directory contains up to 1,024 page table entries, each of which can hold pointers to up to 1,024 pages of physical memory. Together, the two tables can reference 1M pages of 4 KB each, for a total of 4 GB. See Figure 3.4 for an illustration of the paging address translation.

The page table pointer is referred to as the Page Table Entry (PTE), which, in addition to pointing to the physical memory page, contains a number of status bits for the physical page. Included in these is the *present*, or valid, bit that indicates whether or not the referenced page is in physical memory. If the *present* bit is clear, a page-fault is generated to request that the operating system make the page available. See Section "Memory Manager," on page 63 for a complete description of Windows NT memory management.

Also in the PTE is a bit that marks the page as being either user mode or kernel mode and a field that indicates the privilege level necessary to reference the page. Other status bits indicate various conditions of the referenced page, such as whether it has been referenced or modified, if it is read or write protected,

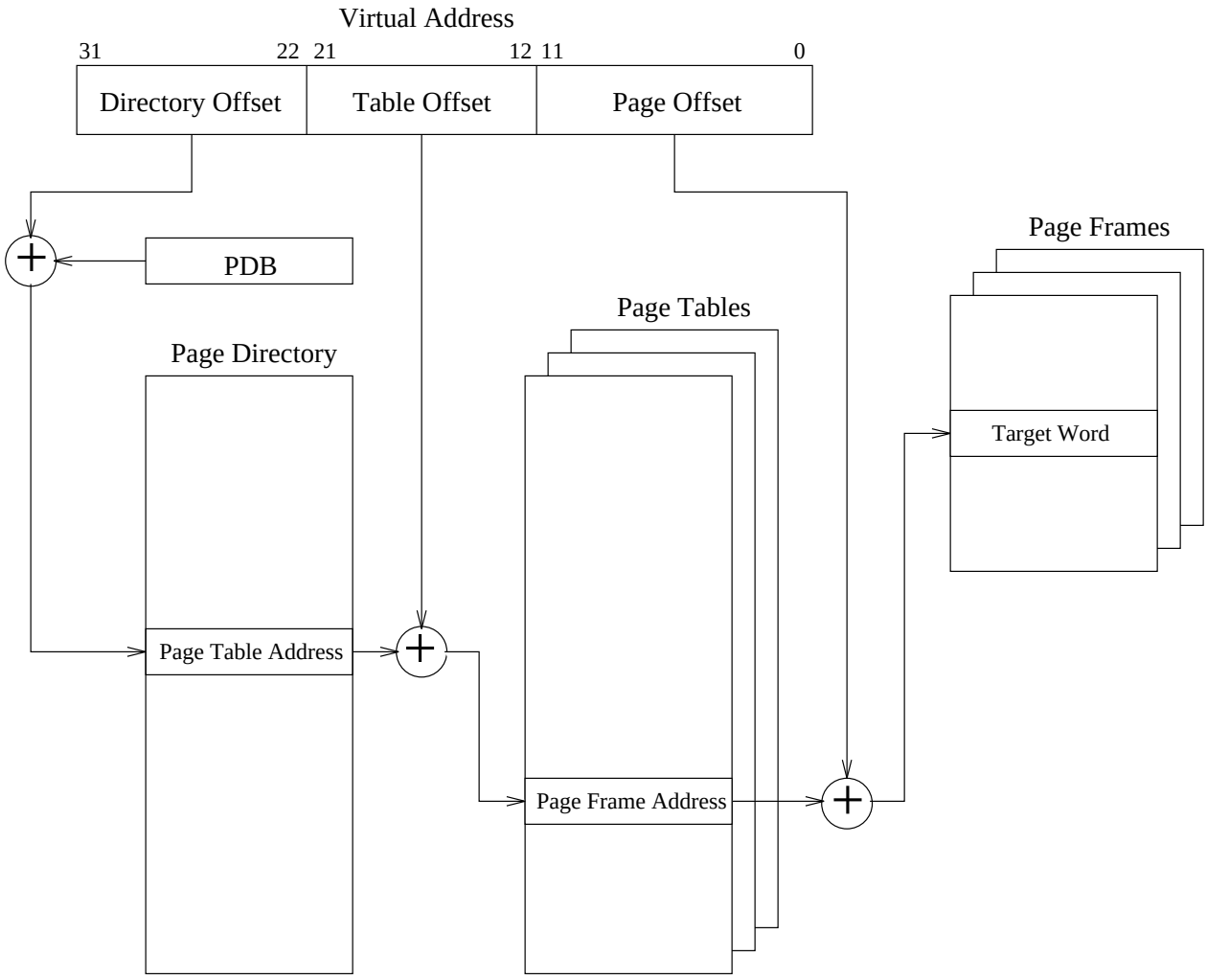


Figure 3.4. Intel x86 Paging Address Translation

and if page caching is enabled. These bits are set by the hardware but are used exclusively by the operating system to implement page protection and demand paging.

3.1.1.5.2 TLB Entries The hardware paging unit translates the linear addresses generated by the segmentation unit into physical addresses. Each internal cache contains a Translation Lookaside Buffer (TLB), which holds the 32 most recently used page translations. The TLB speeds up address translations for recently used pages by preserving the results of the last 32 page resolutions.

The operating system must flush the TLB when entries in the page tables are changed because the old mappings may no longer be valid. The TLB is flushed whenever the CR3 register is loaded. The CR3 register is changed either explicitly (with the MOV instruction) or as a side effect of a task switch. Single entries in the TLB can be flushed using the INVLPG instruction.

Segmentation provides protection by encapsulating regions of memory and by separating users from the kernel. Privilege levels are used to restrict users from accessing the kernel's memory (except under strict conditions via gates). Further, various fields in a segment descriptor define access attributes to a segment (read, write, or execute). Together with privilege levels, these attributes tightly control what operations can be performed on a segment by a process.

3.1.1.6 I/O Support

Input/output is accomplished on the $x86$ through I/O ports, which can be configured for input, output, or both. I/O ports are in an address space separate from main memory and are accessed with dedicated I/O instructions. Both memory and I/O port accesses use the same external bus. The M/IO# processor signal lets external logic distinguish between memory accesses and I/O accesses. I/O address space is only 64 KB in size, so only 16 bits of the address bus is used for an I/O address. It is not segmented or paged. The characteristics of each I/O port are determined by the hardware device that implements it.

I/O instructions provide access to the I/O address space for both configuration and control of port hardware, and data accesses between ports and the processor. The IN and OUT instructions move data and control information between a processor register and a port register in I/O address space. The INS and OUTS instructions are versions of IN and OUT that support efficient looping (with the REP instruction prefix) for the transfer of blocks of data between I/O port registers and memory.

Some I/O devices have access to both I/O space and memory space, and may be set up to transfer blocks of data directly into memory without the processor being involved in transferring every datum. This Direct Memory Access (DMA) mechanism is most often used by block transfer devices like disk drives. Additional DMA implementation details are provided in the sections describing each machine.

Permission is granted to a process to access an I/O port if the process' CPL is less than or equal to the value of the I/O Privilege Level (IOPL) field in the FLAGS register or if the I/O port's bit is clear in the process' I/O Permission Bit Map in its TSS. The IOPL field specifies the lowest privilege level (greatest value of CPL) that the processor will allow to issue I/O instructions without checking the I/O Permission Bit Map. Processes with CPL values greater than the IOPL value that try to access an I/O port whose I/O Permission bit is set will raise a general protection exception.

Windows NT sets the IOPL so that only the executive has unrestricted access to the I/O address space. The I/O Permission Bit Map is used, for example, to give Win32 direct access to video control registers. (Video display memory, however, is mapped to the memory address space and access to it is controlled by the memory protection scheme.)

3.1.2 DEC Alpha AXP Processor Architecture

The DECchip 21064-AA microprocessor is the first implementation of the 64-bit Alpha AXP architecture. It is a RISC processor. As such, its design was driven by this principle: Make frequent operations fast, and provide infrequently needed functions, such as exception handling, in software as sequences of the simple frequent instructions. Hence, exception and protection mechanisms on the 21064-AA are simple, providing as little as possible (but not too little) in hardware for software to build the security features an operating system requires.

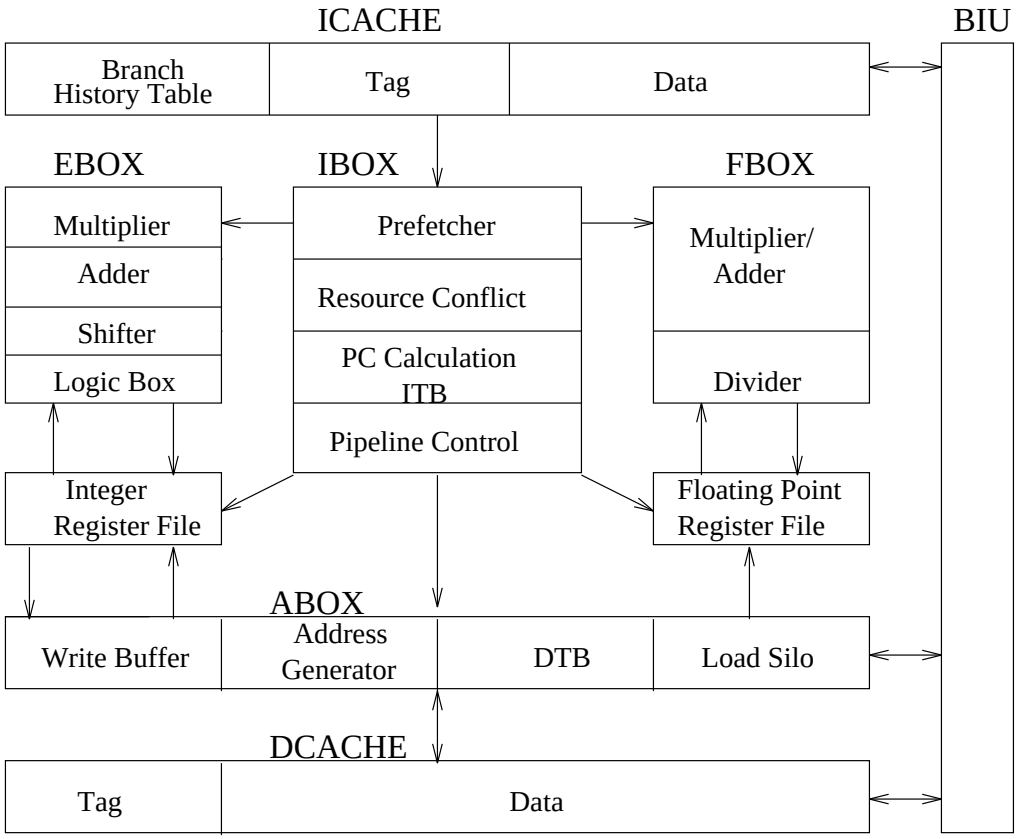


Figure 3.5. Block Diagram of the Alpha 21064-AA

The Alpha AXP architecture uses software, the Privileged Architecture Library (PAL), to implement additional security features, including functions that should be executed only by the TCB. This software is referred to as PALcode and is specific to a particular hardware implementation and operating system.

The description of the processor architecture is divided into two main parts. First, the hardware implementation is described; that is, the functional units, the execution modes, the physical registers, translation buffers, 32-bit addresses, the security-relevant aspects of the instruction set, exception handling, and support for interrupts. Second, the software implementation, including functions and data structures particular to Windows NT and PALcode support for virtual memory management, is described. (See Section "The Privileged Architecture Library for Windows NT," on page 31.)

The section concludes with a description of I/O mechanisms.

3.1.2.1 DEC Alpha AXP Processor Hardware

3.1.2.1.1 Hardware Functional Units The 21064-AA integrates four independent execution units, an 8 KB instruction cache, and an 8 KB data cache (the ICACHE and DCACHE). See Figure 3.5. The four execution units are called the Ibox, Abox, Ebox, and Fbox. The Ibox is the central control unit that issues

instructions while maintaining the instruction pipeline and performing branch prediction calculations. The Ibox contains abort and exception logic, two Instruction Translation Buffers (ITBs), and internal processor registers (IPRs), which are described in Section "Hardware Registers," on page 25. The Abox is the address translation, load/store, and bus interface unit. It contains a Data Translation Buffer (DTB), a four-entry, 32-byte-per-entry write buffer, and IPRs, which are described in Section "Hardware Registers," on page 25. The Abox, through the bus interface unit, can address 16 GB of physical memory. The Ebox is the integer operations unit; the Fbox is the floating point operations unit.

3.1.2.1.2 Execution Modes The Alpha CPU supports four execution access modes: kernel, executive, supervisor, and user (listed in order of most privileged to least privileged). These are used to control access to pages of memory and the use of five privileged machine instructions. (See Section "Privileged Machine Instructions," on page 29.) The processor mode is indicated by the two-bit field in the internal Processor Status (PS) register. (See Section "Hardware Registers," on page 25.) In the PS register's Current Mode field, 00 indicates kernel mode, 01 indicates executive mode, 10 indicates supervisor mode, and 11 indicates user mode. Windows NT only uses two modes, kernel and user. Any machine instruction or software function described as privileged can only be invoked from kernel mode.

Kernel mode allows super page access (described in Section "Translation Buffers," on page 28 but essentially allowing access to either 1 GB or more of non-mapped virtual address space), access to all pages, and use of privileged machine instructions and privileged Windows NT PALcode functions. (See Section "The Privileged Architecture Library for Windows NT," on page 31 for more information about privileged and unprivileged Windows NT PALcode functions.) User mode does not allow access to super pages, allows access to only those pages with the owner bit set to user, and does not allow use of privileged machine instructions or privileged Windows NT PALcode functions.

In addition to these modes there is a special execution mode, called PALmode, that provides an environment for the execution of the PAL functions known as PALcode. PALmode is characterized by the following: instruction stream memory mapping is disabled; all hardware, including internal processor registers, is accessible; and interrupts are disabled.

PALmode is entered and PALcode invoked in one of two ways, either in response to a hardware-detected abnormal event (e.g., floating point error or translation buffer miss) or in response to software issuing the CALL_PAL machine instruction. Depending on the event or the contents of the function field of the CALL_PAL machine instruction, execution is transferred to the appropriate entry point of a PALcode routine. No matter how invoked, every PALcode routine is executed in PALmode.

When PALmode execution completes, the processor returns to *native* mode, with all memory mapping and interrupts enabled and only general purpose registers available to software. This is the mode in which both kernel-mode operating system software and user-mode application software is normally executed.

The execution access modes (kernel, executive, supervisor, and user) and the execution environment mode (PALmode and native mode) identify completely different aspects of the state of the processor. The use of the same word, mode, is unfortunately confusing, but there is no connection between the two types of "mode" implemented in hardware. For example, changing from native mode to PALmode does not automatically change user mode to kernel mode, and changing from user mode to kernel mode does not automatically change native mode to PALmode.

3.1.2.1.3 Hardware Registers The CPU contains 32 64-bit integer registers (R0 through R31) as part of the Ebox and 32 64-bit floating point registers (F0 through F31) as part of the Fbox. The integer registers are entirely general purpose, except for R31, which is always read as zero. Similarly, floating point register F31 is always read as zero. There is a 43-bit Virtual Program Counter (VPC). (The effective virtual address size, a subset of the 64-bit Alpha architecture, is defined by the architecture as a function of the smallest page size for a particular chip implementation; in the case of the 21064-AA, the 8 KB page size determines the 43-bit virtual address.) There are over forty special-purpose IPRs or pseudo-registers. (Pseudo-registers are addresses in the internal register address space with which no storage is associated. Their usefulness is entirely the result of whatever side effects are assigned to their being written to or read from.) There are 32 hardware registers for use by PALcode. For completeness all IPRs are described briefly here; security-relevant registers will be discussed in more detail later.

These registers are part of the Ibox:

- Translation Buffer Tag Register - holds the tag for the next Translation Buffer (TB) update operation; done only in PALmode
- Instruction Translation Buffer Page Table Entry Register (ITB_PTE) - used to write new PTE after an ITB miss; done only in PALmode
- Instruction Translation Buffer Page Table Entry Temporary Register - reads of ITB_PTE require this register as intermediate storage before PTE data is returned to integer register; done only in PALmode
- Instruction Cache Control and Status Register (ICCSR) - contains hardware enables, including the floating point enable bit
- Exception Address Register (EXC_ADDR) - contains virtual address of next instruction after return from exception or interrupt handling; least significant bit indicates if the instruction will be executed in PALmode
- Clear Serial Line Interrupt Register - clears serial line, performance counter, and Correctable Read (CRD) interrupt requests
- Serial Line Receive Register - provides on-chip one-bit-at-a-time serial line input function
- Instruction Translation Buffer ZAP Register (ITBZAP) - write to register invalidates entire ITB; done only in PALmode
- Instruction Translation Buffer ASM Register (ITBASM) - write to register invalidates all ITB PTEs in which the address space match (ASM) bit is not set; done only in PALmode
- Instruction Translation Buffer Invalidate Single Register - write to register will invalidate the single ITB PTE that maps the virtual address in the integer register parameter
- Process Status Register (PS) - contains the two current privilege mode bits
- Exception Summary Register (EXC_SUM) - records various arithmetic traps since last time it was cleared
- PAL Base Address Register (PAL_BASE) - contains the base address of PALcode; reset clears
- Hardware Interrupt Request Register - provides a record of all currently outstanding hardware interrupt requests
- Software Interrupt Request Register - provides a record of all currently outstanding software interrupt requests
- Asynchronous Trap Request Register - provides a record of all currently outstanding asynchronous trap request interrupt requests

- Hardware Interrupt Enable Register - enables up to six different hardware interrupts
- Software Interrupt Enable Register - enables up to fifteen different software interrupts
- AST Interrupt Enable Register - enables up to four asynchronous systems traps
- Serial Line Transmit Register - provides on-chip one-bit-at-a-time serial line output function

These registers are part of the Abox:

- Translation Buffer Control Register - contains two-bit field that selects TB page mapping sizes
- Data Translation Buffer Page Table Entry Register (DTB_PTE) - used to write new PTE after an DTB miss
- Data Translation Buffer Page Table Entry Temporary Register - reads of DTB_PTE require this register as intermediate storage before PTE data is returned to integer register
- Memory Management Control and Status Register (MM_CSR) - records and locks information whenever a data stream fault occurs; software must read Virtual Address register to unlock register for later update
- Virtual Address Register (VA) - records and locks virtual address associated with data stream fault or DTB miss; software must read Virtual Address register to unlock register for later update
- Data Translation Buffer ZAP Register - any write to this pseudo-register invalidates all DTB PTEs
- Data Translation Buffer ASM Register (DTBASM) - any write to this pseudo-register invalidates DTB PTEs with ASM bit clear
- Data Translation Buffer Invalidate Single Register - any write to this pseudo-register invalidates the single DTB PTE that maps the virtual address held in the integer register parameter
- Flush Instruction Cache Register - any write to this pseudo-register flushes the entire instruction cache
- Flush Instruction Cache ASM Register - any write to this pseudo-register flushes all instruction cache blocks with ASM bit clear
- Abox Control Register (ABOX_CTL) - contains bits to enable machine checks on incorrigible errors, interrupt requests on correctable read errors, instruction and data caches, super page mapping (described later), and limited Big Endian support
- Alternate Processor Mode Register - contains two-bit field specifying execution mode to be used for reserved PAL instructions for loading and storing without on-chip memory mapping
- Cycle Counter Register - if enabled, it counts CPU cycles
- Cycle Counter Control Register - one bit enables cycle counter register
- Bus Interface Unit Control Register - controls external cache characteristics, identifies read and write speeds, indicates size of external cache, and turns on and off Error Correcting Code (ECC)

These registers are all read only and used for diagnostics:

- Data Cache Status Register - one bit indicates whether last data stream load or store hit or missed
- Bus Interface Unit Status Register (BIU_STAT) - contains bits indicating various bus interface errors
- Bus Interface Unit Address Register - contains the physical address associated with the errors indicated by the eight least significant bits in BIU_STAT

- Fill Address Register - contains the physical address associated with the errors indicated by bits 8 through 14 in BIU_STAT
- Fill Syndrome Register - if ECC is turned on and an error occurs, it contains the two seven-bit syndromes for the upper and lower 32-bit parts of the bad 64-bit word; syndromes can correct single errors
- Backup Cache Tag Register - contains the results of every backup cache tag probe; if a parity or ECC error occurs, the register is locked and can be read one bit at a time

The 32 PAL_TEMP registers for use by PALcode are accessible only by the privileged machine instructions described in Section "Privileged Machine Instructions," on page 29.

3.1.2.1.4 Translation Buffers There are two instruction translation buffers. One caches eight of the most recently used page translations for 8 KB pages; the other caches four translations for larger pages (64 KB, 512 KB, or 4 MB) specified by the granularity hint field in the page table entry. If the processor is not in PALmode and the page table entry associated with the Virtual Program Counter is in the instruction translation buffer, the IBOX uses the Page Frame Number and the protection bits in the page table entry to check for access violation and complete the virtual to physical address translation for instruction fetches.

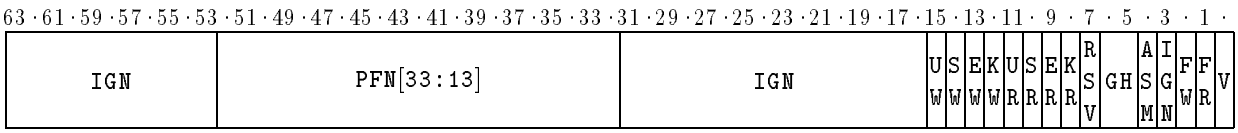
There is one data translation buffer. It caches 32 of the most recently used page translations of data pages from 8 KB to 4 MB as specified by the granularity hint field in the page table entry. If the page table entry associated with the virtual address referenced by a load or store operation is in the data translation buffer, the ABOX uses the Page Frame Number and the protection bits in the page table entry to check for access violation and complete the virtual to physical address translation for data access.

Both translation buffer types support what are called super pages of 1 GB or more of address space. The data translation buffer supports both a large and a small size super page. Super pages can only be accessed in kernel mode. Super pages provide one-to-one virtual to physical address translation without the need for page table mapping. See Section "Windows NT Virtual Address Space," on page 32 for how Windows NT uses the instruction translation buffer's super page mechanism and Section "I/O Support," on page 34 for information about how Windows NT uses the data translation buffer's large super page mechanism.

Both types of translation buffer support the isolation of virtual address spaces with an address space match bit in the page table entry. A write to the ITBASM or DTBASM internal processor registers invalidates all page table entries with the address space match bit not set. This provides an easy mechanism to preserve page table entries mapping operating system code and data while invalidating all other entries.

See Table 3.3 for the field of a DTB page table entry. The ITB page table entry is similar but without write enable fields or fault on read or write fields.

3.1.2.1.5 Hardware Considerations for 32-bit Addresses The 21064-AA is an implementation of a 64-bit architecture. Unlike other RISC architectures (in particular, the MIPS/R4400) there is no 32-bit mode. Operating systems, like Windows NT, that use 32-bit addresses can be written for the architecture because the instruction set includes 32-bit load, store, and arithmetic operations that work on an architecture-specified canonical form for 32-bit data. The canonical form for integers, including 32-bit addresses, is simply the lower 32 bits (31..0) seen by the operating system with the upper 32 bits (63..32) set identical to bit 31.



IGN	Ignore	PFN	Page Frame Number
UW	User Write	UR	User Read
SW	Supervisor Write	SR	Supervisor Read
EW	Executive Write	ER	Executive Read
KW	Kernel Write	KR	Kernel Read
RSV	Reserved	GH	Granularity Hint
FW	Fault on Write	FR	Fault on Read
ASM	Address Space Match	V	Valid

Table 3.3. Translation Buffer Page Table Entry

3.1.2.1.6 Privileged Machine Instructions There are five privileged machine instructions, extensions to the basic Alpha machine instruction set. These can only be issued in kernel mode. Any attempt to execute one of these instructions in user mode results in an illegal reserved or privileged instruction exception. (See Section "Exception Handling," on page 29.) These five additional instructions must be executed in PALmode, unless the hardware enable bit in the ICCSR is set. (See Section "Hardware Registers," on page 25.) If that bit is set, the instructions can then be executed (from kernel mode) in native mode. This bit is never set by Windows NT.

These are the five privileged machine instructions. HW_LD and HW_ST are used to fill and store the on-chip data and instruction translation buffers and data and instruction caches without using the on-chip memory management mechanisms. HW_MFPR and HW_MTPR are used to read and write the internal processor registers. HW_REI is used to return from PALcode to either PALmode or native mode. The EXC_ADDR internal processor register points to the next instruction to be executed after a PALcode routine returns. If the Least Significant Bit (LSB) of the EXC_ADDR is set, execution will continue in PALmode; otherwise, it will return to native mode.

3.1.2.1.7 Exception Handling The 21064-AA enters PALmode when the CPU detects any exception and jumps to the entry point of the appropriate exception-handling PALcode routine. Exceptions are either detected by hardware or initiated by software.

There are four general classes of hardware-detected abnormal event exceptions: reset; hardware exceptions, including machine checks, arithmetic exceptions, illegal reserved or privileged instruction exceptions, and data fetch errors; interrupts, which are described more fully in the next subsection; and TB misses. Each of these events automatically invokes PALmode. The PALcode routines that service these exceptions would be expected to be implemented for any operating system. Hardware-detected exceptions jump to fixed offsets from the base address of the PALcode, which is stored in the PAL_BASE internal processor register. This base address value may be changed, but on reset it is set to zero.

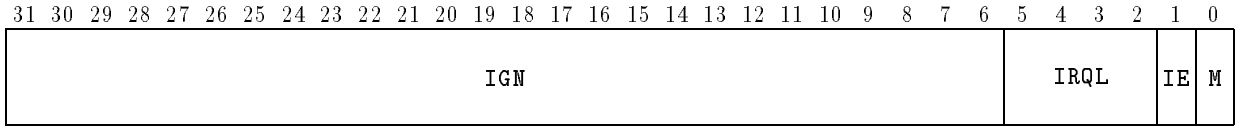
Exceptions that are not detected by hardware are generated by the CALL_PAL instruction. The variety of functions implemented through the CALL_PAL instruction is entirely dependent on the requirements of

the particular operating system for which the PALcode is written. CALL_PAL instructions can be either privileged or unprivileged. CALL_PAL-initiated exceptions jump to entry points determined by the 26-bit function field of the CALL_PAL instruction and the base address of the PALcode. The offset to be added to the base address is determined as followed: if the CALL_PAL is privileged, 0x2000 is added to the lower six bits shifted left six positions; if the CALL_PAL is unprivileged, 0x3000 is added. Bit 7 is clear for privileged CALL_PAL instructions; it is set otherwise. This provides for 64 privileged CALL_PAL entry points to regions of 64 bytes each and 64 unprivileged CALL_PAL entry points to regions of 64 bytes each. Attempts to issue CALL_PAL instructions with privileged function fields from user mode results in a illegal reserved or privileged instruction exception. The same exception also results if the CALL_PAL instruction is issued with the upper 18 bits of the function field not clear.

The 21064-AA entry points for PALcode functions are listed from highest priority to lowest:

- RESET - can occur at any time; most internal processor registers are undefined; but the ICCSR (except for the process number, ASN, and the performance counters, PC0 and PC1), PAL_BASE, ABOX_CTL, and BIU_CTL are cleared; PALcode is expected to initialize ITBZAP, PS, EXC_SUM, SIRR, ASTRR, HIER, SIER, ASTER, SL_XMIT, DTB_CTL, DTBZAP, SL_CLR
- MCHK - caused by uncorrected hardware error
- ARITH - arithmetic exception
- INTERRUPT - includes hardware corrected error
- UNALIGN - addresses must be aligned to a 4-byte value (DEC longword)
- DTB_MISS PAL mode - a data translation buffer miss while in PALcode has to be handled differently than in native mode because of the implied return to EXC_ADDR
- DTB_MISS native mode - handled by PALcode in normal manner
- D_FAULT - PALcode can handle access violation, bad virtual addresses (not in canonical form) and faults on read and write
- ITB_MISS - instruction translation buffer misses are handled by PALcode
- ITB_ACV - instruction translation buffer access violations are handled by PALcode
- CALL_PAL - general (operating system dependent) exceptions are handled by PALcode
- OPCDEC - caused by attempted illegal reserved or privileged instruction use
- FEN - caused by floating point operation when FPE bit is not set in ICCSR or IEEE-defined errors occur

3.1.2.1.8 Interrupts Interrupts are a special category of exception, as listed in Section "Exception Handling," on page 29. Interrupts are asynchronous events that change the flow of execution. The most common interrupts are I/O device requests. The 21064-AA provides six hardware interrupts controlled by pins connected to off-chip sources, 15 software interrupts controlled by the Software Interrupt Request Register (SIRR) internal processor register, and four asynchronous system trap interrupts controlled by the Asynchronous System Trap Request Register. All interrupts are independently maskable by other internal processor registers, i.e., there are no NMIs (Non-Maskable Interrupts, as in the Intel 80x86).



IGN	Ignore
IRQL	Interrupt Request Level (0–7)
IE	Interrupt Enable (irrespective of IRQL)
M	Process Privilege Mode (Kernel or Not-Kernel)

Table 3.4. Processor Status Register

3.1.2.2 The Privileged Architecture Library for Windows NT

Context switching, interrupt and exception handling, and low-level memory management functions are provided by privileged software, residing in RAM, called PALcode. PALcode consists of routines, each identified by an offset from a relocatable base address stored in the internal processor register, PAL_BASE. Each routine is constructed from the basic machine instruction set extended by the five privileged machine instructions described in Section "Privileged Machine Instructions," on page 29. The Alpha AXP is unique among RISC chip designs in its use of software to complete its architecture. The reason for its use by DEC is so the hardware would not be biased in favor of a particular operating system, and thus the porting of the operating systems DEC supports would be easier. It is PALcode that completes the exception and protection mechanisms required to implement security features.

PALcode is specific to a particular combination of operating system and hardware platform and provides functionality normally associated with hardware, microcode, or the read-only BIOS used in personal computers. Thus, the PALcode for Windows NT differs from the PALcode for OSF/1 (a "UNIX-like" operating system) or VMS, the other operating systems DEC supports.

Windows NT PALcode defines 32-bit virtual data structures to work with the operating system, including a virtual processor status register. See Table 3.4 for the layout of the processor status register.

There are privileged Windows NT PALcode functions and unprivileged Windows NT PALcode functions, implemented through the CALL_PAL machine instruction. The former can only be called from kernel mode, while the latter can be called from user mode. If an attempt is made to execute a privileged CALL_PAL function from user mode, an illegal reserved or privileged instruction exception will occur.

Privileged Windows NT PALcode functions include all services the operating system requires, such as enabling/disabling interrupts, maintaining the translation buffers, reading the processor status, returning from an exception, returning from a system call, etc. See Table 3.5 for a list of privileged Windows NT PALcode functions. Unprivileged Windows NT PALcode functions that can be called from user mode include calling system services, requesting debugging services, requesting thread environment information, and enforcing critical regions of code.

Only two of the unprivileged Windows NT PALcode functions do not immediately switch to kernel mode after being called. These functions involve a single process context and are not security relevant. The

Mnemonic	Description	Mnemonic	Description
halt	halt the processor	restart	restart the processor
draina	drain aborts	initpal	initialize the PAL code
wrentry	write system entry	swpirql	swap IRQL
rdirql	read current IRQL	di	disable interrupts
ei	enable interrupts	swppal	swap PAL code
ssir	set software interrupt request	csir	clear software interrupt request
rfe	return from exception	retsys	return from system service call
swpctx	swap privileged thread context	swpprocess	swap privileged process context
rdmces	read machine check error summary	wrmces	write machine check error summary
tbia	translation buffer invalidate all	tbis	translation buffer invalidate single
dtbis	data translation buffer invalidate single	tbisasn	translation buffer invalidate single for ASN
rdksp	read initial kernel stack	swpksp	swap initial kernel stack
rdpsr	read processor status register	rdpcr	read processor control register
rdthread	read current thread value	wrperfmon	write performance monitoring values
rdstate	read internal processor state		

Table 3.5. Privileged Windows NT Call PAL Functions

Address	Permission	Description
0x00000000	User and Kernel	General user address space
0x80000000	Kernel	Non-mapped kernel space (super page)
0xc0000000	Kernel	Mapped, page table space
0xc2000000	Kernel	Mapped, general kernel space

Table 3.6. Virtual Address Map

remaining unprivileged Windows NT PALcode functions switch to kernel mode immediately after PALmode is entered.

3.1.2.2.1 Windows NT Virtual Address Space The 32-bit, 4 GB virtual address space implemented by Windows NT PALcode is divided into 4 regions. The lowest 2 GB is mapped space used in both user and kernel modes for general purposes. The next higher 1 GB is non-mapped, kernel-only space, created and maintained by the hardware-provided super page feature of 21064–AA. The super page addressing extension allows one-to-one mapping of the virtual address to the physical address without TB misses. The third region, immediately above the super page and also kernel-only space, is 64 MB of mapped address space used for page tables. The fourth and highest region is 936 MB of mapped, general kernel space. All mapping is done through the TBs. See Table 3.6 for layout of the virtual space.

3.1.2.2.2 Windows NT Virtual Address Translation An Windows NT virtual address using an 8 KB page size consists of two fields: a 19-bit virtual page number and a 13-bit offset. See Table 3.7 for the two fields of a virtual address.

Virtual addresses are translated to physical addresses via the TBs. If the address is not in canonical form, or if the valid bit is not set in the PTE, or if the process mode is user and the owner bit is clear (which indicates kernel mode), or if the access is a write and the dirty bit is clear, an exception occurs and PALmode

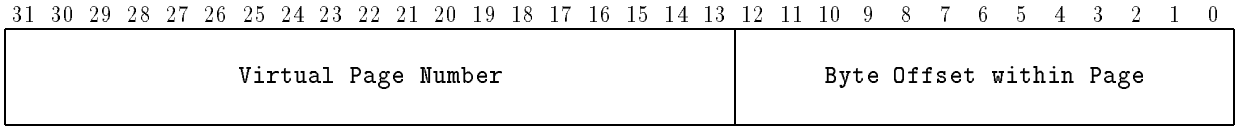
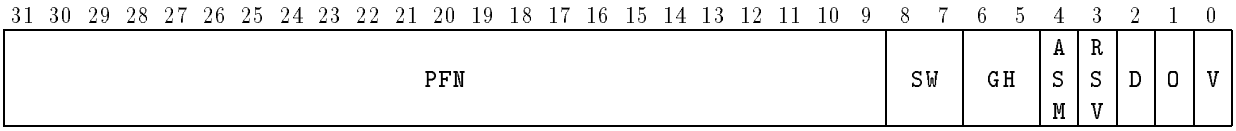


Table 3.7. Virtual Address



PFN	Page Frame Number	SW	Reserved for OS
GH	Granularity Hint	ASM	Address Space Match
RSV	Reserved	D	Dirty flag
O	Owner flag	V	Valid flag

Table 3.8. Page Table Entry Fields

is entered. Windows NT PALcode maintains and fills all TBs. The virtual address associated with the exception is stored in the VA internal processor register.

For TB misses Windows NT PALcode implements a virtual to physical memory mapping scheme similar to that implemented in hardware by Intel 80x86 processors. Each process has its own pointer to the base of its page directory (PPDR). On a TB miss the Windows NT PALcode routine is jumped to with this argument and the virtual address. The virtual page number of the address is interpreted as two fields, an 8-bit Page Directory Index and an 11-bit Page Table Index. These two fields select the page table and the page frame number within the page table. The other 13-bit field indicates the byte within the page.

3.1.2.2.3 Windows NT PAL Page Table Entries and Memory Protection A 32-bit Windows NT PAL virtual PTE consists of the following fields: a 23-bit Page Frame Number (PFN), two bits reserved for the operating system (SW), a two-bit Granularity Hint (GH), a 1-bit Global or Address Space Match (ASM), an owner bit, a dirty bit, one bit reserved, and a valid bit. The PFN points to a physical 8 KB page frame. The GH field provides a way to map to a multiple of the standard page size; the multiple is 1, 8, 64, or 512. The ASM bit indicates that the translation is global to all processes, and therefore does not have to be invalidated when a process switch occurs. This provides a simple hardware mechanism to preserve entries that map operating system regions while invalidating all others. The dirty bit is initially clear. The first write causes a fault, and the Alpha-specific part of the Memory Manager examines one bit in the SW field to see if the page is writable. If it is, the memory manager sets the dirty bit to indicate the page has been written. If the bit indicates the page is read only, an invalid access exception occurs. The owner bit is set to 0 for kernel and 1 for user. The valid bit is set to 1 if the PFN is valid. See Table 3.8 for the layout of a PTE entry.

3.1.2.3 I/O Support

Only one hardware feature of the 21064-AA is used to support I/O: super pages. This feature is enabled by setting a two-bit field in the ABOX_CTL internal processor register. There are two types of super pages, large and small. Windows NT I/O on Alpha platforms uses the large super page mechanism to provide one-to-one virtual to physical address translation when the highest two virtual address bits (42..41) are set equal to two. Super page translation is only allowed in kernel mode, and thus only the Windows NT executive has unrestricted I/O address space access.

There is no other support for I/O provided by the 21064–AA. There are no I/O ports and no special I/O instructions as on the Intel x86 family of processors. I/O reads and writes are simply reads and writes of memory locations in the special address space described above. (See Table 3.11 for how this space is mapped to different I/O devices.)

I/O is performed by assembly language macros accessed through the Hardware Abstraction Layer (HAL) of Windows NT. Addresses for registers of common PC I/O devices like floppy disk FIFO controllers or VGA controllers are converted by the Windows NT executive to what are called Quasi-Virtual Addresses (QVA). These addressed are passed to the HAL access macros, where they are converted to 34-bit physical addresses that will be put on the MBUS. If the highest two bits of the physical address are not equal to zero, the address is destined for the HBUS; otherwise it is an on-board memory address, not an I/O address.

The HBUS is essentially a clone of the pin interface of the EISA bus used in IBM PC clones. Thus, the physical address must contain two bits that identify the type of address (on-board memory, local I/O, EISA memory, or EISA I/O) and four bits to determine the four pin byte enable setting on an Intel 386 (or above). This leaves 28 bits remaining for potentially 32-bit EISA addresses. Hence, a Host Address Extension register on the system board, as on EISA systems, is used to provide the upper 7 bits of the EISA address if necessary.

So, for example, to create a 34-bit physical address that targets a EISA I/O device, bits 33..32 would be set to three, bits 8..7 are set to determine the byte offset, bits 6..5 are set to the byte length, and bits 24..2 of the target address are placed in bits 31..9 of the physical address to be put on the bus. (Since the HBUS always uses 32-bit addresses, the lower two bits of the target address (1..0) cannot be put on the bus, i.e., they are assumed zero and the byte enable bits are necessary to determine exactly which bytes of the fetched word to use.) All this address manipulation is done by the HAL access macros. For more details about I/O address space details and the functions and relationships of different buses on the Alpha platform, see Section “DECpc AXP/150,” on page 44.

3.2 Evaluated Machine Architectures

Windows NT is designed to be a portable operating system that can be “shrink wrapped” and sold for use on a variety of commercially available servers, Personal Computers (PCs), and workstations, based on a variety of processors. The previous section provided an overview of the processors which are used in evaluated machines. This section provides an overview of the HAL, which Windows NT uses to encapsulate the differences between machines which use the same processor, and an overview of each of the evaluated machines.

3.2.1 Hardware Abstraction Layer

This section describes the purpose and philosophy of the HAL, the types of functions that it provides, and its role in making a single copy of Windows NT portable across all computers which use a specific processor.

The differences among the computers on which Windows NT runs are divided into those that originate with the choice of Central Processing Unit (CPU), and those that result from the choices made in designing a machine around the CPU. The first arise from processor-specific features and the latter from machine-specific features. Note that, for this discussion, the boundary between CPU and machine is not necessarily coincident with the CPU chip boundary. For example, the Windows NT architecture specifies both first- and second-level memory caches as features of the machine rather than the processor, even though all evaluated processor chips have internal first-level caches.

Processor-specific functionality is embodied in a few modules in the Windows NT executive (the Microkernel, described in Section "Microkernel," on page 49, and the Memory Manager, described in Section "Memory Manager," on page 63). The HAL encapsulates all machine-specific functionality. It is implemented as a DLL and so is not bound to Windows NT until runtime, after distribution and installation.

As a result of this design, porting Windows NT to a computer from one which uses the same CPU requires only the replacement of the HAL, whereas porting to a computer which uses a different CPU may require changes to the Microkernel and Memory Management code, will require a recompilation of all of Windows NT, and will require a different HAL. This is convenient, since variations in implementation are more common, especially in the personal computer world, than variation in processor.

The HAL also provides a uniform interface for device drivers, which use the HAL's generic functions to access hardware devices. This allows device drivers to also be machine independent and portable.

HAL's abstraction of machine features also frees the designers of PC compatible machines from rigid conformance to the dated Personal Computer/Advanced Technology (PC/AT) design details, encouraging the creative optimization of PC platforms. Machine implementation can be freely changed so long as commensurate changes are made to the HAL. This design renders that portion of Windows NT outside the HAL, 95–98% portable on first-time ports to a new processor (although requiring a recompilation), and 100% portable to machines which use a previously ported processor (and no recompilation is required).

The set of functions included in the HAL depends upon whether various functions are performed by hardware (often the case with CISC processors) or by software (often the case with RISC processors). However, the set of HAL functions for a given processor is constant across all implementations.

The machine features which HAL abstracts include hardware buses (system buses, I/O buses, and multiple instances of buses), DMA controllers, interrupt controllers, system timers, cache coherency mechanisms, cache flushing, and symmetric multiprocessor support. Most of these abstractions are common to all HALs.

- Bus Support. The HAL manages multiple buses and translates any logical bus address to a system usable address. The HAL manages access to bus configuration information (e.g., EISA, PCI) and must know how to find configuration information on all relevant buses. It also maps bus interrupts to the system interrupt descriptor table.
- DMA Support. The HAL manages access to the system DMA controller. It abstracts the system memory model for I/O functions, including: providing per-transfer access to the DMA, initiating DMA transfers, providing full memory scatter/gather data transfer, and accommodating address-space

incompatibilities between hardware adapters and the computing system during DMA transfers (often required for older hardware with limited DMA addressability).

- Interrupt Support. The HAL manages the priority of interrupts. It abstracts the hardware mechanisms for interrupts, which vary by processor and machine. For example, the Alpha processor handles interrupts directly, whereas, the HAL handles interrupts on the x86 machines. The HAL abstracts hardware priorities by assigning different device interrupts to different interrupt request levels (Interrupt Request Level (IRQL)) (see Section "I/O Subsystem," on page 88).

- Multiprocessor Support. The HAL provides specific support to multiprocessor systems by abstracting the ability to start multiple processors, to initialize multiple processors and processor control structures, and to send interprocessor interrupts to specified processors. The HAL supports hardware-based dynamic interrupt routing by maintaining an IRQL for each processor and sending incoming interrupts to the processor with the lowest IRQL.

 The HAL insulates the Microkernel from the administrative details of controlling multiple processors, such as: how to interrupt a processor, how to start a processor, and how to reboot a processor.

- System Timer Support. The HAL provides a consistent Executive interface to a periodic clock capability. The HAL calls the Microkernel at periodic intervals from each processor (for performance profiling), and queries a performance counter for the fine-grain timings needed by device drivers.

- Miscellaneous Support. The HAL provides support for a number of other functions including: system boot, system failure, powerfail support, and returning to firmware for reboot.

3.2.2 Evaluated Machines

The evaluated configuration consists of the Microsoft Windows NT Workstation and Server Version 3.5 operating system running on any of the following machines:

- Compaq Proliant 2000 or 4000 Server (Intel Pentium processor)
- DECpc AXP/150 (DEC ALPHA processor)

These systems are available in various configurations of disk storage, removable media, peripherals, memory, and display. Memory capacities range from 16 MB to 512 MB. The user has many choices among CRT display types, keyboards, and printers. Some of the evaluated machines were designed to be network file servers. However, for this evaluation they are configured as stand-alone workstations without network connections.

3.2.2.1 Compaq Proliant 2000 and 4000

The Compaq Proliant 2000 and 4000 are both built around Compaq's Tri-Flex architecture. The Proliant 2000 has sockets for two processor modules and the Proliant 4000 has sockets for four processor modules. The 4000 cannot accomodate internal hard disk drives in the main chassis, so an auxilliary chassis is provided.

Two types of processor modules are evaluated, differing only in speed: Model 5/90-1 uses a 90 MHz Pentium processor and Model 5/100-1 uses a 100 MHz Pentium processor. A block diagram of the Compaq Proliant architecture is provided in Figures 3.6 and 3.8. The Tri-Flex buffer is a set of bus latches in two Super Data

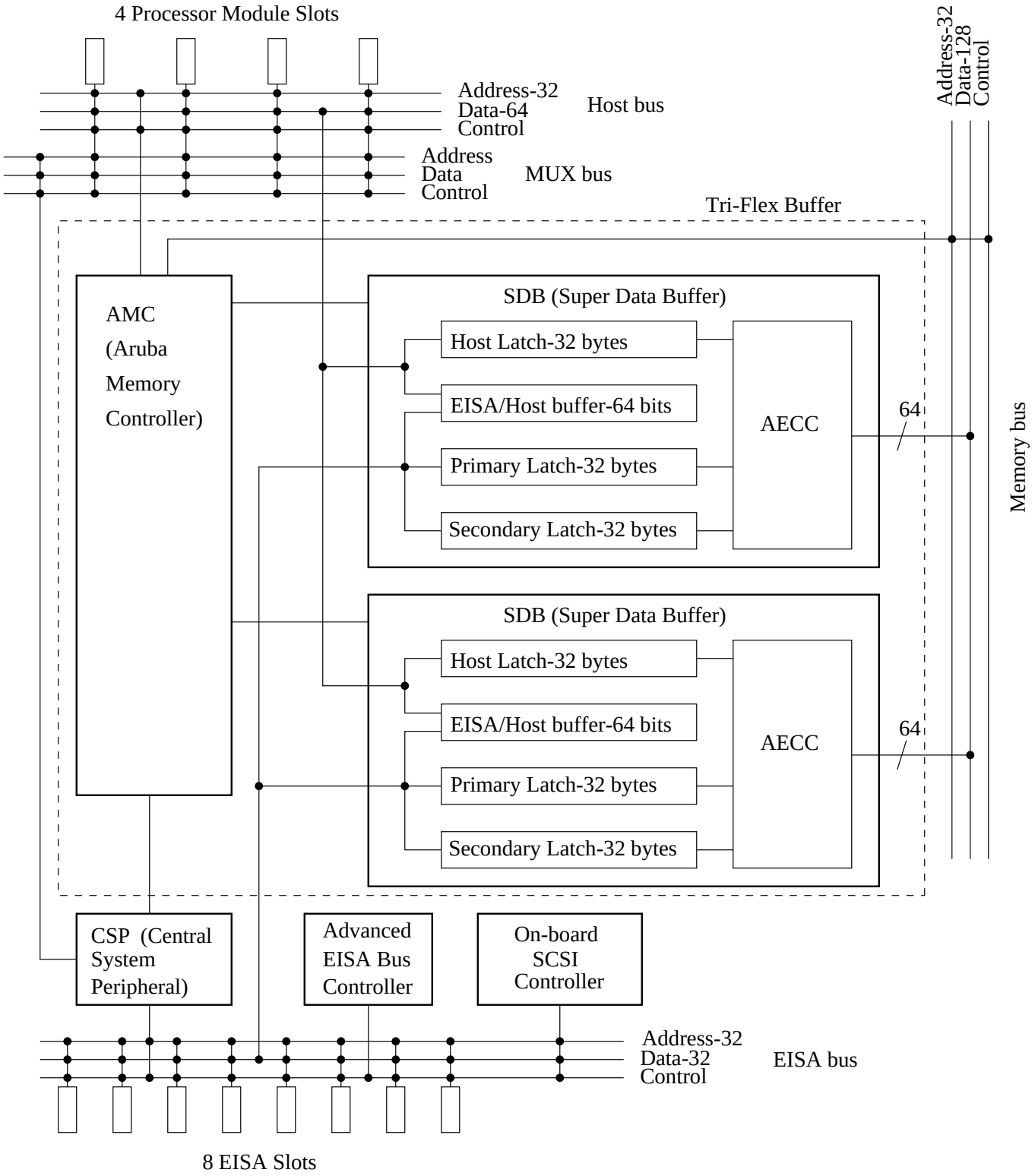
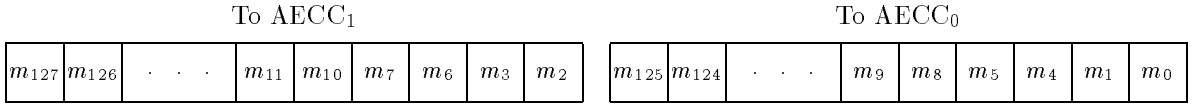


Figure 3.6. Compaq Proliant Main System Board



Routing of the 128 data lines (m_0 through m_{127}) from the Memory bus to the two AECCs so that the complete failure of any 4-bit Dynamic Random Access Memory (DRAM) module (e.g., one providing bits $m_0, m_1, m_2,$ and m_3) can be corrected. Each AECC is capable of correcting two consecutive bits.

Figure 3.7. Memory Bit Routing for AECC 4-Bit Protection

Buffer (SDB) chips and an Aruba Memory Controller (AMC) chip which interface three of the four main system buses. Each of these buses has a different width, operates at a different clock rate, and has a different control protocol. The Tri-Flex buffer interfaces these different buses to provide for the efficient transfer of data among them and to allow the buses to operate asynchronously. The four main buses are:

- The **host bus**, which has slots for one to four processor modules. It is 64 bits wide and runs at half the processors' external speed (25 or 33 Megahertz (MHz)).
- The **memory bus**, which has sockets to accept system main memory chips. It is 128 bits wide and has an error correcting mechanism which can correct single-bit and some multi-bit errors (see later discussion of Advanced Error Correcting Code (AECC) in this section).
- The **Extended Industry Standard Architecture (EISA) bus**, which has eight industry standard EISA slots to accept I/O controllers and other system peripherals. It is 32 bits wide and runs at 8.33 MHz. Each slot can accept a 32-bit EISA card or a 16-bit or 8-bit Industry Standard Architecture (ISA) card.
- The **MUX bus**, which distributes 16-level hardware interrupts, DMA and NMI status, and software resets to the processor modules. It also provides access to some Distributed System Peripheral (DSP) registers via the EISA bus. The MUX bus is not directly interfaced to the Tri-Flex buffer.

3.2.2.1.1 The Main System Board The Compaq Proliant main system board contains slots for from one to four processor modules (all processor modules in a machine must run at the same speed), a memory subsystem containing from 16 MB to 512 MB of dynamic Random Access Memory (RAM), and an interface to an eight-slot EISA bus. The Intel Pentium processor architecture is described in Section "Intel Pentium Processor Architecture," on page 11. These major components of the Proliant are interconnected by the Compaq Tri-Flex buffer.

The AMC chip receives data requests from the system processors and the EISA bus masters during each cycle. It determines if the requester is a system processor on the Host bus or a peripheral processor on the EISA bus, and whether the request is for main memory or an EISA card. It then sends address and control signals to the SDB, directing it to deliver the requested data to the appropriate bus.

Memory contents are protected by an AECC mechanism, capable of detecting and correcting both single-bit and adjacent double-bit errors within a 64-bit word. This requires an extra bit of storage per byte (a standard feature of the PC architecture, although it is used in the PC only for parity checking).

In order to protect all 128 bits of data on the memory bus, the bus is logically divided into two 64-bit buses, each fed to a separate AECC. To allow the correction of what is deemed the most likely multi-bit memory failure (the complete failure of a Single Inline Memory Module (SIMM) DRAM module—which supplies a nibble, or four adjacent bits), the buses are split in such a way that half of each DRAM nibble provides adjacent bits in each 64-bit half of the bus, thereby allowing a whole DRAM-aligned nibble to be corrected by the pair of AECCs (see Figure 3.7).

All main system boards have built-in Small Computer Standard Interface (SCSI) controllers. They may also be configured with an additional EISA SCSI controller card.

The SCSI controller built into the Proliant main system board uses the NCR 53C710 advanced SCSI controller chip, and is implemented as a full EISA bus master. It can connect up to seven internal and external SCSI devices and has a maximum transfer rate of 10 MB per second.

Control of the EISA bus is arbitrated by the Compaq proprietary Application Specific Integrated Circuit (ASIC) Central System Peripheral (CSP) chip. Potential bus masters are organized into four classes: system main processors, EISA bus masters, the DRAM refresh controller, and the DMA controller. Control of the bus is first arbitrated at the class level and assigned to one class on each arbitration cycle. Within the chosen class, control is then arbitrated among the devices of that class that have made requests. This prevents two devices in the same class from becoming bus master consecutively while devices from other classes are waiting. Any device, at any time, even those not capable of mastering the bus, may request that the current bus master be pre-empted and that a new arbitration cycle begin.

The Compaq proprietary AMC chip controls arbitration on the Host bus. It selects the one of those processors that are requesting a bus cycle that has not had a cycle the longest, regardless of which processor has had its current request outstanding the longest. This limits the impact of processors that are using the bus heavily on the performance of lower-use processors.

The AMC is the only master on the memory bus, so no arbitration is necessary.

Hardware interrupts from EISA cards are detected by the CSP and forwarded across the MUX bus to the DSP chip on each processor module. The DSP controller is a Compaq proprietary ASIC which contains two Intel-8259-compatible interrupt controllers and two Intel-8254-compatible timers. The interrupt controller portion of each DSP forwards interrupts to its processor. The processor determines the source of the interrupt and executes the appropriate interrupt handler.

Windows NT allows any processor to handle any interrupt. Interrupt assignment is machine specific, and may be fixed in hardware or assigned by software. The Proliant has software-assignable interrupts, which are assigned by the HAL. The HAL always enables interrupts 1, 3, 4, 5, 6, and 12 for processor 0; and it enables interrupts 0 (system timer), 8 (profiler), and 13 (inter-processor interrupt) for all processors.[2] The other interrupts (except 2) are assigned to processors according to the interrupt number modulus the number of processors (e.g., interrupt 11 on a four processor system is assigned to processor three). Interrupt 2 is ganged with 9 and follows the modulus rule for 9 instead. The resulting interrupt distribution for systems with 1 to 4 processors are as shown in Table 3.9.

The CSP controller is a Compaq proprietary ASIC which contains two Intel-8237-compatible four-channel DMA controllers. The controllers are cascaded in typical EISA bus fashion. That means that one controller

[2] Interrupts which are enabled for all processors are independently activated for each processor (i.e., they are not electrically ganged). Each processor generates its own system timer and profiler interrupts, which target only itself. The inter-processor interrupt targets specific processors via system board logic.

Interrupt	1 Processor	2 Processors	3 Processors	4 Processors
0 - System timer	0	01	012	0123
1 - Keyboard	0	0-	0--	0---
2 - I/O channel	0	-1	0--	-1--
3 - COM2	0	0-	0--	0---
4 - COM1	0	0-	0--	0---
5 - Fixed drive	0	0-	0--	0---
6 - Diskette drive	0	0-	0--	0---
7 - LPT1	0	-1	-1-	---3
8 - Profiler	0	01	012	0123
9 - I/O channel	0	-1	0--	-1--
10 - Reserved	0	0-	-1-	--2-
11 - Reserved	0	-1	--2	---3
12 - Mouse	0	0-	0--	0---
13 - Inter-processor	0	01	012	0123
14 - Fixed drive	0	0-	--2	--2-
15 - Hard drive	0	-1	0--	---3

Table 3.9. Interrupt Distribution in Multi-Processor Compaq Systems

is designated the master and one the slave, and that transfers over all of the slave's four channels must share the use of one of the master's channels. This provides seven DMA channels between the two controllers.

The keyboard controller is an Intel 8251 microcontroller with its own dedicated firmware in on-chip Read Only Memory (ROM).

The Proliant has re-programmable configuration memory and Basic I/O Supervisor (BIOS) ROM. The Proliant machines also have a re-programmable Test ROM. Re-programmable means that the user may employ Compaq supplied software to replace the BIOS or Test code.

The system configuration is stored in Non-Volatile Random Access Memory (NVRAM) on the main system board. A configuration maintenance program is supplied with the system to change the configuration data in this NVRAM. The configuration of both standard and optional features may be stored here and changed with the maintenance program. Optional features which can be configured include those on EISA expansion cards made by Compaq and other manufacturers.

3.2.2.1.2 Processor Modules The Compaq Proliant system accepts processor modules that plug into connectors on the Host and MUX buses. Two types of processor modules are in the evaluated configuration: a 90 MHz Intel Pentium module and a 100 MHz Intel Pentium module. All processor modules in a particular system must run at the same speed. A system can be configured with one to four processor modules.

Each system processor module contains the following components:

- A processor chip (90 or 100 MHz Pentium),
- A 512 KB second-level memory cache and cache controller,

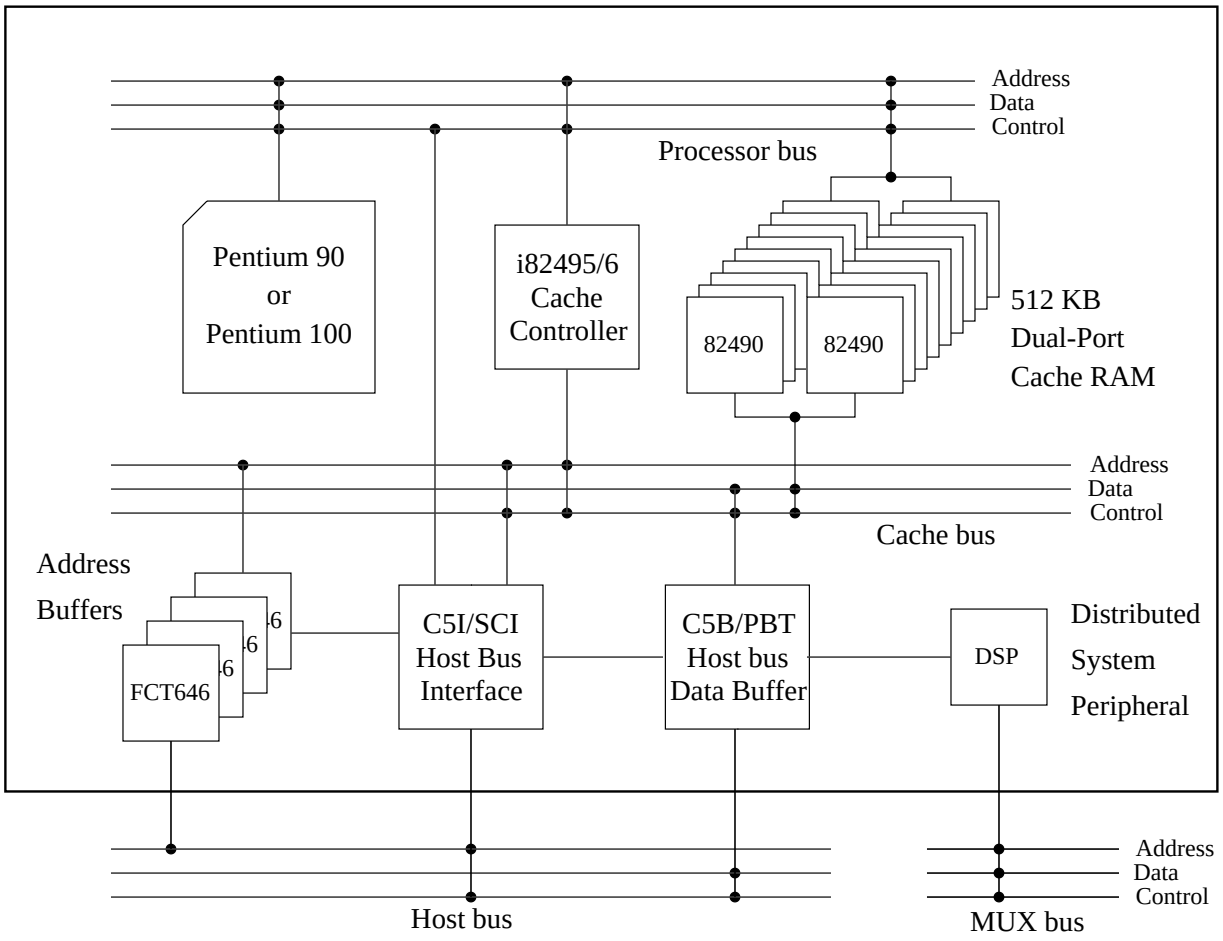


Figure 3.8. Compaq Proliant Processor Module

- Host bus address buffers,
- A host bus interface,
- A host bus data buffer, and
- A Distributed System Peripheral (DSP) controller.

Each processor module has one processor chip, either a 90 MHz or 100 MHz Intel Pentium. The processor contains control and floating point units and separate 8 KB data and instruction first-level caches. The only substantial difference between the processors is speed.

The on-chip first-level caches are described in Section "Intel Pentium Processor Architecture," on page 11. The second-level cache (on the processor module) is implemented with an Intel 82496 cache controller and 82491 cache RAMS, which were designed to integrate with the first-level Intel processor caches.

Consistency between the first-level cache on the processor chip and the second-level cache on the processor module is assured by an interfacing policy implemented by the cache controllers that maintains the first-level

Model		Processor	EISA HD Controller	Standard Disk Capacity	Hot[a] Disk Bays
Proliant 2000	5/90-1	Pentium/90	SMART	External	5
	5/100-1	Pentium/100	SMART	External	5
Proliant 4000	5/90-1	Pentium/90	SMART	External	7
	5/100-1	Pentium/100	SMART	External	7

Table 3.10. Compaq Model Features

[a]Hard disk drives can be installed in and removed from hot drive bays without opening the case and without powering the system down. Drives may only be installed in and removed from internal drive bays when the power is off, and then only by opening the case. The Proliant system supports an external expansion chassis with up to seven hot drive slots.

caches as a subset of the second-level cache. Address snooping signals are propagated throughout the system so that all caches on all processor modules remain consistent. The second-level cache and the first-level data cache in the Pentium both implement the full Modified/Exclusive/Shared/Invalid (MESI) cache coherency protocol. The Pentium's first-level instruction cache, which is read-only, implements a more efficient subset of the MESI protocol.

There are provisions for selectively disabling the caches by memory page, for choosing the update mechanism (write-through or write-back) by memory page, and for locking cache entries to memory for certain instruction types. The latter supports processor synchronization mechanisms such as gates and semaphores, which require atomic read-alter-rewrite cycles.

The processor module's Cache bus is interfaced with the main system board's Host bus by the Compaq proprietary ASIC chips: SCI, PBT, and the FCT646s. The SCI directs cache line fills, main memory writes, and cache snoops to the Host bus; and sends processor I/O cycles to the MUX bus. The FCT646s buffer address lines between the Cache bus and the Host bus. The PBT chip buffers data lines between the Cache, Host, and MUX buses, and passes address and control data to the MUX bus, all under control of the SCI.

3.2.2.1.3 Standard Configurations The Proliant 2000 and 4000 include the following components:

- $3\frac{1}{2}$ inch 1.44 MB floppy diskette drive,
- Dual speed CD-ROM drive,
- Main memory of 64 MB,
- Video display system,
- Keyboard,
- Mouse.

3.2.2.1.4 Power-On Password The Proliant's hardware configuration may be set to trigger a firmware power-on password mechanism that prevents the system from powering-up until a password is correctly entered from the console keyboard. The trigger and the password are stored in NVRAM and are routinely changed by use of a Compaq-supplied configuration editor that runs only under MS-DOS. The configuration cannot be changed before the power-on password check is passed. The password validation trigger can be

reset without knowing the previous password only by setting a switch on the main system board, which requires opening the (lockable) system box.

3.2.2.1.5 Boot from Removable Media The system hardware configuration stored in NVRAM can be set to disable booting from any removable media device (e.g., a floppy drive). The system can be configured to prevent changing this setting without knowing the power-on password. However, if access is available to the inside of the system box,[3] the power-on password can be reset, as explained in Section "Power-On Password," on page 42.

3.2.2.1.6 Protection of Date and Time Setting The date and time may be changed by modifying the system configuration stored in NVRAM, using the MS-DOS `DATE` and `TIME` commands, using the Windows NT `DATE/TIME` window on the Control Panel, or using the `DATE and TIME` commands in the Windows NT Virtual DOS window. The Windows NT date and time changing functions both require administrator privileges to execute successfully. The NVRAM can be protected by the method described in Section "Protection of System Configuration," on page 43, which also prevents the use of MS-DOS commands.

However, if access is available to the inside of the system box, the date and time can be changed by booting to MS-DOS from a floppy and either running the Compaq hardware configuration editor or using the MS-DOS `DATE` and `TIME` commands.

3.2.2.1.7 System Case Locking The Proliant has a key lock that prevents the case from being opened and also locks a door on the front of the case which prevents the hot drives from being removed. The system will not power up when the case is open, although this can be defeated. The system will run while it is unlocked and the case is closed.

3.2.2.1.8 Protection of System Configuration Compaq provides a hardware configuration editor that runs only under MS-DOS. A user could also write an MS-DOS program to change the hardware configuration.

The system can be configured to prevent booting to MS-DOS by disabling boot from removable media and not installing an MS-DOS partition on any hard drive. Once this configuration has been established, subsequent changes to the configuration can only be made by opening the system case, resetting the configuration to the default state (which allows booting from the floppy drive), booting MS-DOS from a floppy diskette, running the Compaq configuration editor, restoring the system configuration to its correct setting, (including the required updates) and rebooting Windows NT. In the course of making each configuration change, the administrator must remember to return the configuration to a state that prevents booting from floppy.

Users running an MS-DOS emulation session under Windows NT cannot edit the hardware configuration because they lack the necessary access to the real I/O space of the NVRAM. Windows NT provides no functionality for modifying the hardware configuration.

[3] The Trusted Facilities Manual (TFM) requires that the system box be locked.

34-bit Address			Address	
33	32	31–0 Range	Block Name	Description
0	0	0000.0000-007F.FFFF	Local Memory	On-board RAM (128 MB)
0	1	0000.0000	Local I/O	EISA INTA
0	1	8000.0000-8003.FFFF		$FEPROM_0$ (256 KB)
0	1	A000.0000-A00F.FFFF		$FEPROM_1$ (1 MB)
0	1	C000.0000-C1FF.FE00		Combo Chip (64 KB PC I/O Map)[a]
0	1	D000.0000		HAE Register
0	1	E000.0000		SYSCTL Register
1	0	0000.0000-FFFF.FFFF	EISA Memory	EISA RAM (4 GB)
1	1	0000.0000-FFFF.FFFF	EISA I/O	EISA I/O (4 GB)

Table 3.11. Alpha AXP Memory Map

[a]Bits 8–0 contain byte counts and byte shifting instructions.

3.2.2.2 DECpc AXP/150

The DECpc AXP/150, henceforth called the AXP/150, is a machine built around DEC's Alpha Reduced Instruction Set Computer (RISC) processor, but conforming as much as possible to the standard PC architecture. A system block diagram is provided in Figure 3.9.

3.2.2.2.1 Main System Board The AXP/150 main system board, system model number PB22H-KB, consists of one DEC Alpha 21064-AA processor running at 150 MHz, a 512 KB second-level memory cache, a memory subsystem, several local buses, and an EISA bus with six slots. The DEC Alpha 21064-AA processor is described in Section "DEC Alpha AXP Processor Architecture," on page 23.

3.2.2.2.1.1 Memory Mapping 34-bit addresses are used on most buses. The two most-significant bits of the 34-bit addresses carried throughout the main system board are used to divide the address space into four areas: Local Memory, Local I/O, EISA Memory, and EISA I/O. A 1 in address bit 33 causes the access cycle to be routed to the EISA bus, a 0 causes it to reference an on-board device. A 1 in address bit 32 distinguishes between I/O accesses and memory accesses (see Table 3.11). The other 32 bits specify an address in the 4 GB address range selected by the upper two bits. Less significant bits may be used to map devices to portions of each block's address space.

3.2.2.2.1.2 Memory Subsystem The memory subsystem consists of the memory control logic and main memory. The memory control logic is integrated with the cache control logic and is responsible for generating memory addresses and the relevant control signals that allow memory accesses to complete. Two banks of 128-bit wide memory are mounted directly on the system board. The machine can support memory configurations from 16–128 MB in increments of 16 MB. This memory is addressed in the Local Memory block.

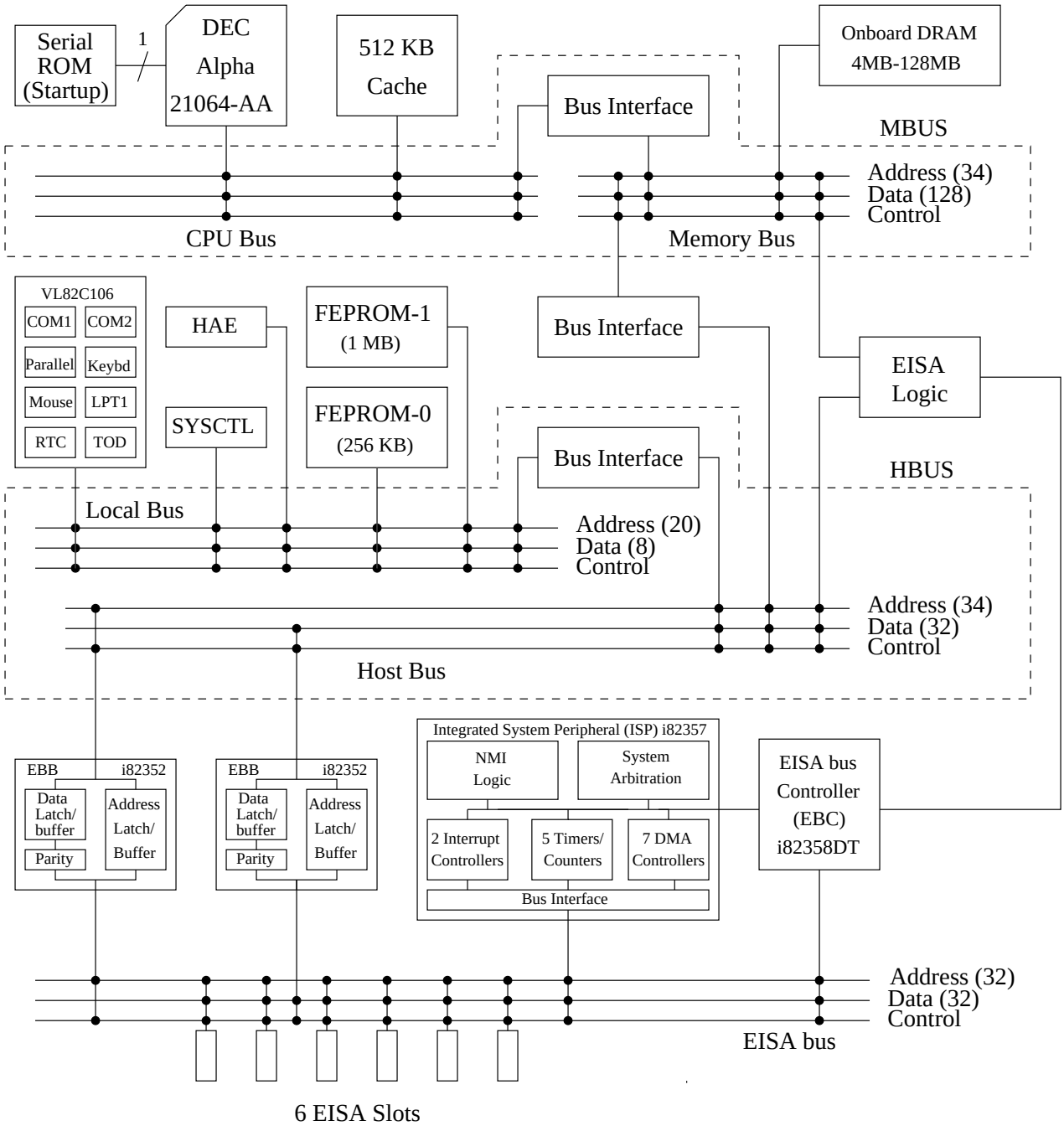


Figure 3.9. DECpc AXP/150 System Block Diagram



Figure 3.10. SYSCTL and HAE Registers

3.2.2.2.1.3 Memory Cache The DEC Alpha 21064-AA has separate on-chip 8 KB first-level instruction and data caches, as described in Section "DEC Alpha AXP Processor Architecture," on page 23. Storage for a 512 KB second-level write-back cache with 32-byte lines is provided on the system board. PALcode routines control the second-level cache and interface it to the first-level cache. The processor snoops the MBUS during cycles controlled by other bus masters in order to keep the caches consistent with memory content. Only addresses in the Local Memory block (i.e., bits 33–32 = 00) are cached.

3.2.2.2.1.4 System Buses There are five buses on the main system board. These are grouped into three logical buses according to the range of addresses to which their devices respond.

- The **MBUS** consists of the CPU and Memory buses, and is directly driven by the CPU. All addresses are propagated over this bus, but components on the MBUS (on-board DRAM) only respond to local memory addresses. MBUS addresses are 34 bits wide, and the MBUS data path is 128 bits wide.
- The **HBUS** consists of the Host and Local buses. It connects all on-board peripheral components and the EISA interface logic to the MBUS. HBUS components respond to Local I/O addresses. On-board peripherals include two ROMs, the Host Address Extension (HAE) and System Control Register (SYSCTL) registers, the VTI 82C106 Combo chip, and the interface to the EISA bus. The Host part of the HBUS carries 34-bit addresses and has a 32-bit data path. The Local part carries 20-bit addresses and has an 8-bit data path.

 The HBUS is normally controlled by the CPU. During DMA, however, the CPU is forced off the HBUS and the HBUS is controlled by the Integrated System Peripheral (ISP). During EISA Master-to-Host Memory cycles, the HBUS is controlled by the EISA interface logic.
- The **EISA bus** supports six industry-standard EISA slots, each of which accept 32-bit EISA cards or 16-bit or 8-bit ISA cards. All add-on peripheral cards connect to the EISA bus. The EISA bus responds to EISA memory and EISA I/O addresses. It runs at 8.33 MHz, carries 32-bit addresses, and has a 32-bit data path.

The HAE register contains the high-order seven bits (31–25) of addresses destined for the EISA bus, as in Figure 3.10.

The SYSCTL register contains two 2-bit memory size codes (one for each SIMM bank) and single-bit flags to control front-panel LED indicators.

3.2.2.2.1.5 The EISA Chipset The EISA bus controller is built from chips in the Intel 82350DT EISA chip set. From this set, the AXP/150 uses one i82358DT EISA Bus Controller (EBC), two i82352 EISA Bus Buffers (EBBs), and one i82357 ISP.

The EBC is the central component of the EISA system. It translates CPU bus cycles into EISA bus cycles and generates EISA bus control signals for DMA transfers, EISA and ISA bus master cycles, and main processor EISA cycles. It also performs any required byte swapping and assembly (to interface among the EISA bus' 8-, 16-, and 32-bit memory paths and devices on other local buses) and controls the routing of data between memory and the EISA bus. Masters on any of the three buses communicate with the other buses through the EBC.

The Intel EBB chip provides latches for the buffering of data or address lines between the synchronous EISA bus and the MBUS. It can also generate and check parity on its channels. One EBB is used in the AXP/150 to implement the data swap logic. The other is used as an address buffer.

The ISP combines many EISA support components into a single device, including a seven-channel i8237A compatible 32-bit DMA controller, five i8254 compatible 16-bit counter/timers, two i8259 compatible eight-channel interrupt controllers ganged in standard PC/AT fashion to provide 15 hardware interrupt levels, NMI generation and control logic, DRAM refresh generation logic, and EISA bus arbitration logic.

3.2.2.2.1.6 Combo Chip On-board I/O devices are controlled by the VLSI Technology 82C106 ISA Combo chip, which supports two serial ports with full modem control, a parallel port (used for control of front panel indicators and the detection of switch positions), real-time and time-of-day clocks, and printer, mouse, and keyboard ports. The chip also has 64 bytes of battery-backed memory, 14 bytes of which is reserved for clock control registers and 50 bytes for use in storing system configuration data. The battery-backed memory may be reset to system default values by temporarily shorting two pins, allowing the recovery from configuration errors which render the system inoperable.

3.2.2.2.2 Standard Configuration The following is the minimum hardware configuration required to run Windows NT.

- 16 MB RAM (4 each 4 MB ME524-DE SIMMs),
- 245 MB RZ24L hard disk
- $3\frac{1}{2}$ inch 1.44 MB RX26 floppy diskette drive,
- RRD42 CD-ROM drive,
- Adaptec AHA-1742a SCSI adapter,
- SVGA graphics adapter and monitor,
- PS/2 keyboard and mouse.

3.2.2.2.3 Startup The reset circuitry in the Alpha CPU is designed to pre-load its instruction cache from an on-chip serial line and then execute the instructions in the cache. The AXP/150 supplies this initial cache load over a 9600 baud line between the CPU and a serial ROM. Once loaded, the serial ROM code configures the Alpha's memory access logic, then tests the integrity of the boot code stored in the Flash Erasable Programmable Read Only Memory (FEPROM), and, if successful, transfers to the boot code. If errors are found, control is passed to a mini-console implemented in the pre-loaded Serial Read Only Memory (SROM) code.

The boot code loads the Alpha's PALcode and triggers a PAL machine reset (a simulated hardware reset to the virtual PAL machine). It then builds memory data descriptors and page tables (after surveying installed

memory). Next, it configures system devices, performs its self-tests, and transfers to the startup console menu routine.

The startup menu provides the user with options to change the system and EISA configurations, set the date and time, select the disk partition from which to boot, or boot from the selected partition. The last option is automatically executed if another option is not chosen within a short time.

3.2.2.2.4 Power-On Password Systems with Firmware version numbers equal to 3.5-5 or greater (required of all systems in the evaluated configuration) can be configured to require that a password be supplied before the system will boot or allow any console input. This allows the system administrator to protect the hardware configuration from unauthorized modification. The administrator can thereby prevent unauthorized booting from floppy diskette and the configuration of any hard drive attached to the external SCSI port.

3.2.2.2.5 Boot from Removable Media The system can be configured so that it will not boot from the floppy drive. It must also not be configured with hard drives that can be replaced without opening a lockable system box (either the main or an auxiliary system box). Hard drives or other devices that might be attached to the external SCSI port on the back of the cabinet will not be able to be configured without first entering the power-on password.

3.2.2.2.6 Protection of Date and Time Setting A menu of functions offered on the console at boot time includes a date and time change feature. This menu is not offered until after the power-on password has been entered successfully. Once Windows NT is running, the date and time setting is protected. The integrity of the date and time is important in maintaining accurate audit logs.

3.2.2.2.7 System Case Locking The AXP/150 has a key lock that prevents the case from being opened. The system will run while the case is unlocked.

3.2.2.2.8 Protection of System Configuration The protection of the integrity of the Windows NT operating system requires that the configuration of the hardware be controlled (including the setting of the date and time) and all other operating systems (including another copy of Windows NT) be prevented from being installed or booted. Instructions for accomplishing this are provided in the TFM. Once this configuration has been established, subsequent changes to the configuration can only be made by the system administrator.

Users running an MS-DOS emulation session under Windows NT cannot edit the hardware configuration because they lack the necessary access to the real I/O space of the NVRAM. Windows NT provides no functionality for modifying the hardware configuration.

Chapter 4
Software Architecture

Figure 2.1, on page 6, depicts the software structure of the Windows NT TCB. This chapter first describes the executive portion of the Windows NT TCB, followed by a discussion of the protected servers in Section "Protected Servers," on page 106, and administrator tools in Section "Administrator Tools," on page 128.

4.1 Executive

The Windows NT executive includes the Microkernel, Object Manager, Virtual Memory Manager, Process Manager, Security Reference Monitor, Local Procedure Call (LPC) Facility, I/O subsystem (which includes the file systems, I/O Manager including the Cache Manager, and device drivers), Configuration Manager, and the Executive Object Services subsystem. Each of these subsystems are discussed in this section.

Each subsystem exports a set of services, called executive system services (system calls). These services are available outside of the executive. Each subsystem also exports a set of interfaces available only within the executive. The Windows NT executive is discussed in detail in this section.

4.1.1 Microkernel

The Microkernel is the lowest layer of the Windows NT executive, and provides services only to the rest of the executive (see Figure 2.1 on page 6). The Microkernel performs the most fundamental operations in Windows NT, determining how the operating system uses the processor or processors, specifically for interrupt handling and scheduling. Basically, the Microkernel serves as a layer between the rest of the executive and the HAL and provides processor architecture-specific support. There is one Microkernel for all $x86$ uniprocessor machines, one Microkernel for all $x86$ multiprocessor machines, and one Microkernel for all DECpc AXP/150 machines. The Microkernel is different from the rest of the Windows NT executive in several ways.

The Microkernel is never paged out of memory and, although it can be interrupted to execute an interrupt service routine, its execution is never preempted. The Microkernel always runs in the privileged mode of the processor. The Microkernel performs the following functions:

- Thread scheduling and context switching
- Exception and interrupt handling
- Low-level multiprocessor synchronization
- Other processor-specific functions such as TLB flush and initialization

The Microkernel exports a set of *microkernel objects* to the rest of the Windows NT executive that control central processing and support the creation of executive objects. Most executive objects encapsulate one or more kernel objects. The Microkernel does not export any objects directly to user mode; objects exported by the Microkernel are encapsulated in one form or another by portions of the executive prior to being exported to user mode (see Section "Executive Object Services," on page 104). There are two types of kernel objects: *dispatcher objects* and *control objects*. This section discusses these objects in detail as well as the differences in the Microkernel for each of the architectures in the evaluated configuration.

4.1.1.1 Dispatcher Objects

Dispatcher objects control scheduling and synchronization and have a *signaled* state (allowing a thread to suspend on the object while waiting for the signaled state to change). When the signaled state is set, the dispatcher object is said to be signaled. When the signaled state is clear, the dispatcher object is said to be not signaled. Each dispatcher object handles its signaled state differently, as described below. The dispatcher objects include:

- Mutant
- Event and event pair
- Semaphore
- Timer
- Thread
- Process
- Queue

4.1.1.1.1 Synchronization Objects A *mutant* is a mechanism for mutual exclusion. A mutant is owned by a thread (see the next section for a discussion of threads). As long as a thread holds a mutant, the thread does not receive any normal Asynchronous Procedure Calls (APCs) (see Section "Asynchronous Procedure Call Interrupts," on page 55); only special APCs are received. When the owning thread releases the mutant, the mutant goes into the "signaled" state, and one of the threads waiting on the mutant is resumed by the Microkernel. The resumed thread acquires the mutant and returns the mutant to the "not signaled" state. A mutant is exported by the Executive Object Services subsystem, however, the exported mutant is called a *mutex* (see Section "Executive Object Services," on page 104).

There are two kinds of events: a *notification event* and a *synchronization event*. The initial state of a notification event is selected when the object is created. When the event occurs, its state becomes "signaled" and it remains "signaled" until it is explicitly cleared. One or more threads can be resumed when the notification event becomes "signaled." A synchronization event works the same way except that only one thread will be resumed when the synchronization event becomes "signaled," regardless of how many threads are waiting on the event. An event is exported to user mode via the Executive Object Services subsystem (see Section "Executive Object Services," on page 104).

A *semaphore* is a counter that provides control of critical resources. As long as the semaphore count is non-zero, the semaphore is in the "signaled" state. A thread trying to acquire a semaphore that is "not signaled" (i.e., its count is zero) will wait until the semaphore is "signaled" (i.e., non-zero). If the semaphore is "signaled," then the thread decrements the count and continues. When the thread releases the semaphore, the count is incremented. A semaphore is exported to user mode via the Executive Object Services subsystem (see Section "Executive Object Services," on page 104).

A *timer* is set for a time interval. When the timer expires it is in the "signaled" state. All threads waiting on the timer are resumed when the timer is "signaled." A timer is exported to user mode via the Executive Object Services subsystem (see Section "Executive Object Services," on page 104).

An *event pair* is a special dispatch object used for context switching between two user-mode threads, such as the Win32 protected server (see Section "Win32," on page 118) and its client applications. A thread holds an event pair, consisting of an event to signal and an event to wait on. To do a context switch to the server, for example, the client signals one half of the event pair and waits on the other half. A context switch is done to the server, the server performs the necessary functions, and then signals the other half of the event pair and waits on the first. The server signals the same half of the event pair that the client is waiting on, so the client can resume as soon as the context switch is complete. An event pair is exported to user mode via the Executive Object Services subsystem (see Section "Executive Object Services," on page 104).

A Microkernel *process* object consists of an address space, a set of threads, and a signaled state. When the signaled state is "not signaled," the process is active; when the signaled state is "signaled" the process is terminated. The Microkernel process object has no knowledge of handles so it bypasses the object table and points directly to the Microkernel threads, the page table directory, the total time threads have executed, the default base priority, and the processor affinity.[1] A process is exported to user mode via the Process Manager (see Section "Process Manager," on page 81).

A Microkernel *queue* object has a list of threads waiting on the queue (which may be none), a current concurrency count, and a maximum target concurrency count. A queue object can be initialized, have entries inserted on the list, and have entries removed from the list. When an entry is inserted on the queue, a waiting thread (i.e., the thread that has been waiting the longest) will be awakened as long as the concurrency count is less than the maximum target concurrency count, which is established during initialization of the queue. If a thread attempts to remove an entry from an empty queue, the thread is placed in a wait state.

4.1.1.1.2 Threads A *thread* is the execution agent in Windows NT. A *Microkernel thread* object is contained within an executive thread object (see Section "Process Manager," on page 81) and represents the only information the Microkernel needs in order to dispatch the thread for execution. A Microkernel thread has a register context which defines the machine state of the thread, a base and current priority for scheduling purposes (priorities are discussed in detail later in this section), an affinity (in multiprocessor systems) which identifies the set of processors on which the thread can run, and one of the following scheduling states (shown in Figure 4.1):

- *Initialized* - The Process Manager has allocated space for an executive thread object and has called the Microkernel to initialize the thread object contained within it (see Section "Process Manager," on page 81).
- *Ready* - The thread is eligible to be run on a processor; it is waiting to execute.
- *Standby* - The thread has been selected to run next on a particular processor and will be run at the next context switch. Only one thread can be in the standby state for a processor at a time.
- *Running* - The thread is executing on a processor. Execution continues until its time quantum expires, it is preempted by the Microkernel to run a higher priority thread, it voluntarily enters the waiting state, or it terminates.

[1] In an MP system, the set of processors on which a thread may execute.

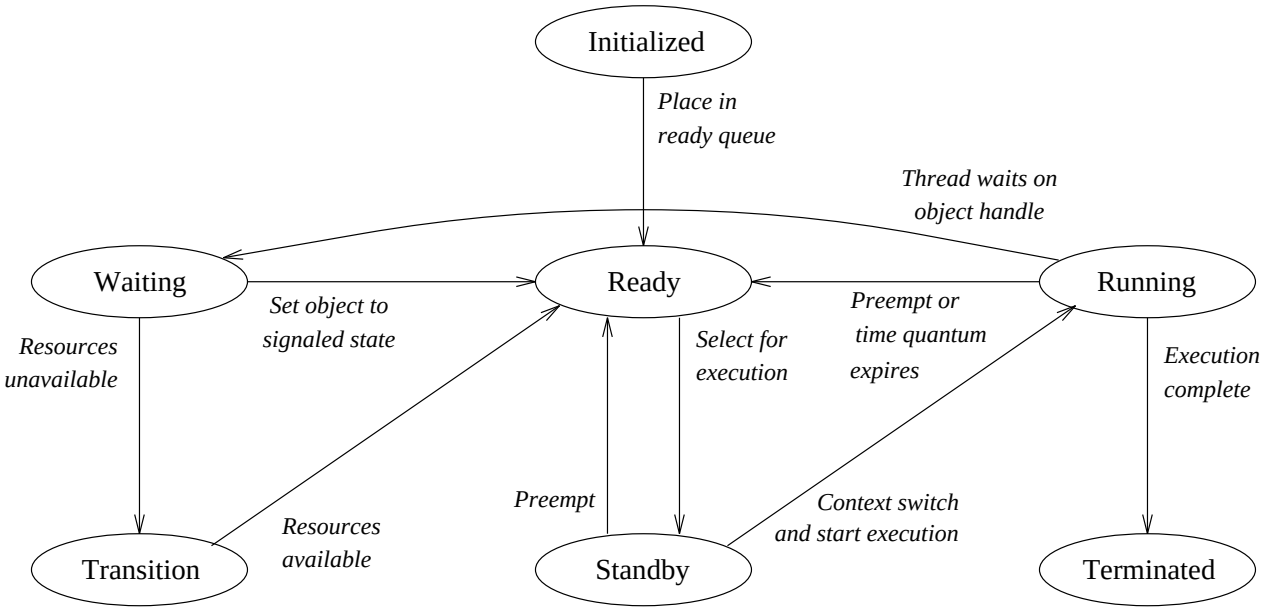


Figure 4.1. Thread States

Priority Level Type	Number of Levels
Real time	16
Variable	15
System	1
Idle	1

Table 4.1. Priority Levels

- *Waiting* - A thread can wait voluntarily on an object for synchronization purposes; the executive can wait on the thread's behalf; or a protected server can direct the thread to suspend itself. The thread returns to the ready state for rescheduling when the wait ends.
- *Transition* - A thread enters the transition state if it is ready for execution but the resources it needs are not available. For example, the thread might have come out of a wait state but the Microkernel stack is not yet in memory. When resources are ready, the thread enters the ready state.
- *Terminated* - When a thread finishes executing, it enters the terminated state. A thread is in a "not signaled" state as long as it is active and goes into its "signaled" state as soon as it terminates. A terminated thread may be deleted or reinitialized by the executive and reused.

Initially, a thread gets a priority when it is created called the *base priority* that indicates the lowest priority level to which it will ever be reduced. There are 32 priority levels in Windows NT as depicted in Table 4.1.

The 16 real-time priorities are not commonly used in Windows NT. Real-time priorities are used for critical kernel-mode threads such as the *modified page writer* (see Section "Physical Memory Management," on

page 67). Within the real-time priorities, thread priority does not change. Scheduling is preemptive, priority scheduling, i.e., a thread at a higher priority can preempt a thread at a lower priority. Round-robin scheduling is done within a priority level only. In other words, if a thread gets to a quantum end (the time allocated for it to execute ends), another thread within its priority level is scheduled using a round-robin algorithm.

Within the 15 variable priority levels, a thread's priority can be raised or lowered. When a thread completes a wait and becomes ready to run, it gets a small boost in its priority and runs at this higher priority until its quantum ends or it waits again. At that point, its priority is reduced back to its base priority. A common problem is called *dynamic deadlock* in which a high priority thread needs a resource, a low priority thread has the resource, and a middle priority thread is compute-bound so the low priority thread doesn't get to run. To solve this problem, the Microkernel examines some subset of scheduled threads every 2 seconds and significantly boosts the thread's priority for 1 quantum. In this way, if a dynamic deadlock situation exists, it can be broken.

One priority level, the system priority level, is reserved for the executive, which uses this priority level to initialize free pages. This way, a ready supply of initialized, free pages always exists. Threads other than the *zero page writer* thread do not compete to run at this level. The last priority level is for idle threads and is not considered part of the 32 priority levels. When a processor has no other threads to run, it runs the idle thread until another thread becomes ready.

When a thread is posted to the ready queue, the dispatching algorithm is executed by the posting processor to attempt to assign a processor to the thread. The dispatcher selects the processor that last executed the newly posted thread, if it is idle; else it selects an idle processor, if there is one; else it selects the processor that last executed the thread, if it is currently running a lower priority thread. If none of these conditions are met, the thread is left in the ready queue and the posting processor returns to its current thread. If the posting processor is selected, it saves its current state and dispatches to the new thread. If an idle processor is selected, the new thread is left in the ready queue for the idle processor to find,[2] and the posting processor returns to its current thread. If a busy processor is selected, the new thread is left in the ready queue, an interrupt is sent to the busy processor, and the posting processor returns to its current thread. The algorithm is designed to ensure that the highest priority thread is always running (but not that the n highest priority threads are running) and, to the extent possible, that processors are assigned to the threads they last executed (to minimize memory cache flushes, TLB flushes, and FPU register transfers between processors).

The dispatching algorithm is the same for single and multiple processor machines, although a stripped down implementation of it, without the unneeded multi-processor components, is provided for single processor microkernels in order to reduce dispatching overhead.

The FPU scheduling algorithm differs by number and type of processor. For single-processor x86 machines, the FPU context is saved and restored on demand, rather than on context switches. When a thread loses a processor, the FPU is locked (by setting the TS bit in CR0), causing the next attempt to use it to generate a Coprocessor-Not-Available interrupt (INT 7). The interrupt handler determines which thread's state is loaded in the FPU and changes state only if necessary. This eliminates nearly all FPU state saves and restores in the common case where only one thread in a system is using the FPU.

For multi-processor x86 machines, if the FPU has been used by a thread its state is saved along with the CPU state during a context switch, but it is only restored on demand (i.e., when a Coprocessor-Not-Available

[2] Idle processors run kernel idle-threads, which do nothing but search the ready queue for threads to start.

interrupt is received and the current thread's FPU state is not already loaded). This prevents having to retrieve FPU state from the FPU of a processor that is currently running another thread.

For RISC machines (regardless of the number of processors) the FPU context is saved and restored along with the rest of the CPU contents during context switches.

Windows NT implements a symmetrical multi-processing scheme in which interrupts can be processed by any processor that receives them. Some machine architectures do not support the distribution of interrupts to multiple processors. Of the evaluated machines, only the Compaq supports multiple processors and it does allow the distribution of interrupts among them. The distribution scheme is explained in Section "The Main System Board," on page 38.

4.1.1.2 Control Objects

Control objects are passive objects used for executive and device driver control. Control objects do not have a signal state and are not waitable. Control objects include:

- Interrupts
- Device queues
- Profiles
- APCs
- Deferred Procedure Calls (DPCs)

An *interrupt* object is used by device drivers. A device driver connects to an interrupt object so that when the interrupt occurs the driver gets control. Interrupt and exception handling is discussed in Section "Interrupt and Exception Handling," on page 55.

Device queues are also used by device drivers to queue IRPs as they progress through the various stages of I/O processing as described in Section "I/O Subsystem," on page 88.

Profile objects are used to profile information within the Microkernel such as address ranges and buffers used for tracing.

An *APC* object is a Microkernel object that describes an APC interrupt. APC interrupts are discussed in Section "Asynchronous Procedure Call Interrupts," on page 55. APC objects are not exported outside of the executive.

DPC objects are Microkernel objects that are not exported outside the executive. A DPC object is used within the executive especially by device drivers. The most important piece of information that a DPC object contains is the address of the routine that the Microkernel will call when it processes the associated DPC interrupt. DPC interrupts are discussed in Section "Deferred Procedure Call Interrupts," on page 54.

4.1.1.2.1 Deferred Procedure Call Interrupts A DPC performs an executive function, other than the function of the current thread. The functions are called "deferred" because they might not execute immediately, however, they will execute before any further thread execution. DPCs execute only after the Microkernel, or often a device driver, finishes more important work and lowers the processor's IRQL below dispatch/DPC level (IRQLs and dispatch/DPC level is discussed below). DPCs provide the executive with the capability to generate an interrupt and execute an executive function in kernel mode. The Microkernel

uses DPCs to process timer expiration, and to reschedule the processor after a thread's time quantum expires. Device drivers use DPCs to complete I/O requests.

A DPC is represented by a DPC object that contains the address of the executive function that the Microkernel will call when it processes the DPC interrupt. DPC routines that are waiting to execute are stored in a DPC queue managed by the Microkernel. To request a DPC, the executive calls the Microkernel to initialize a DPC object and then places it in the DPC queue. Unlike APCs discussed below, there is only a single DPC queue. This prompts the Microkernel to request a software interrupt at dispatch/DPC level because DPCs are generally queued by software running at a higher IRQL and the Microkernel runs the DPC at a lower IRQL. When the IRQL drops, the DPC routine executes without regard to which thread is running, which is often a user-mode thread since they execute at low IRQLs. Since a DPC could execute in the address space of a user-mode thread, a DPC cannot acquire executive resources or modify the borrowed thread's virtual memory. A DPC can call Microkernel functions but cannot call executive system services, generate page faults, or create or wait on objects. Only the executive can queue a DPC.

4.1.1.2.2 Asynchronous Procedure Call Interrupts An *APC interrupt* allows the Microkernel to break into the execution of a *specific* thread and direct it to execute a procedure. All threads (user-mode and kernel-mode) queue APCs to a particular thread by calling the Microkernel; i.e., user-mode threads cannot queue an APC directly, an APC can only be queued via the Microkernel.

APCs queued from a kernel-mode thread can interrupt any target thread and execute a procedure without the target thread's intervention or consent. Kernel-mode APCs do not require permission from the target thread.

An APC queued from a user-mode thread can only target a thread that has declared itself *alertable*. A user-mode thread can declare itself to be alertable by waiting on an object handle and specifying that its wait is alertable or by testing directly whether it has a pending APC.

APCs waiting to execute reside in an APC queue, managed by the Microkernel. The APC queue is thread-specific, each thread has its own APC queue. When asked to execute an APC, the Microkernel will insert the APC object into the queue belonging to the thread that will execute the APC. To process an APC, the Microkernel requests a software interrupt at APC level (which is a low IRQL level), interrupts the target thread's current execution, and the target thread begins executing the procedure at the address indicated in the APC object. The Microkernel resumes the target thread's execution when the APC completes.

Unlike DPCs, since an APC executes within the context of a particular thread at a lower IRQL, the APC can acquire resources, wait on object handles, incur page faults, and call system services.

4.1.1.3 Interrupt and Exception Handling

The Windows NT Microkernel supports both interrupts and exceptions. An *interrupt* is an asynchronous event that can occur at any time, unrelated to what the processor is executing. Interrupts are generated primarily by I/O devices, processor clocks, and timers, and can be enabled or disabled.[3] An *exception* is a synchronous condition resulting from the execution of a particular instruction and can be reproduced. Memory access violations, certain debugger instructions, divide-by-zero errors, and system service calls are all examples of exceptions.

[3] Interrupts that indicate bad memory or a bus error can generally not be disabled.

IRQL	Type of Interrupt
High Level	Machine check or bus error
Power Level	Power failure
Interprocessor Interrupt Level	Work request from another processor
Clock Level	Interval clock
Device Level n	Highest priority I/O device
...	...
Device Level 2	Second Lowest priority I/O device
Device Level 1	Lowest priority I/O device
Dispatch/DPC Level	Thread dispatching and DPC processing
APC Level	APC processing
Low Level	Normal thread execution

Table 4.2. Interrupt Request Levels (IRQLs)

4.1.1.3.1 Interrupts The Microkernel uses an interrupt object to represent interrupts. An interrupt object contains the address of the Interrupt Service Routine (ISR) and other information needed to resolve the interrupt. In the $x86$ architecture, the Processor Control Block (PRCB) contains a pointer to a hardware data structure called the Interrupt Dispatch Table (IDT). When the processor is interrupted, it queries the controller to get the actual interrupt vector and index into the IDT to get the address of the interrupt dispatch routine. In the DECpc AXP/150 architecture, the Interrupt_Entry register holds the address of the interrupt dispatch routine. Control is then transferred to the interrupt dispatcher to handle IRQL synchronization and call the ISR with the appropriate context to resolve the interrupt.

The *interrupt dispatcher* responds to interrupts. Device drivers supply interrupt service routines (ISRs) to service device interrupts and the Microkernel provides interrupt handling routines for other types of interrupts. The interrupt dispatcher maps hardware interrupt levels onto a standard set of *IRQLs* recognized by the Microkernel. The IRQLs are ranked by priority which is different than the scheduling priorities previously described.[4] A thread running in kernel mode on a processor can raise or lower the IRQL setting of the processor on which it is running to mask or unmask lower-level interrupts. Table 4.2 shows the IRQLs and the mapping between the hardware IRQL and the type of interrupt, which is performed by the interrupt dispatcher. Each processor's IRQL setting determines which interrupts that processor can receive. As a kernel-mode thread runs, it raises or lowers the processor's IRQL. Interrupts from a source with an IRQL above the current setting interrupt the processor, while interrupts from sources with IRQLs equal to or below the current level are blocked, or masked.

4.1.1.3.2 Exception Handling When an exception occurs, the processor *traps*, which means it switches from user mode to kernel mode, transferring control to a fixed location in the Microkernel, called the *trap handler*.

The trap handler disables interrupts briefly while it records the machine state and creates a *trap frame* in which it stores the state of the processor at the time the trap occurred. The trap frame contains the following information:

[4] A scheduling priority is an attribute of a thread while an IRQL priority is an attribute of an interrupt source.

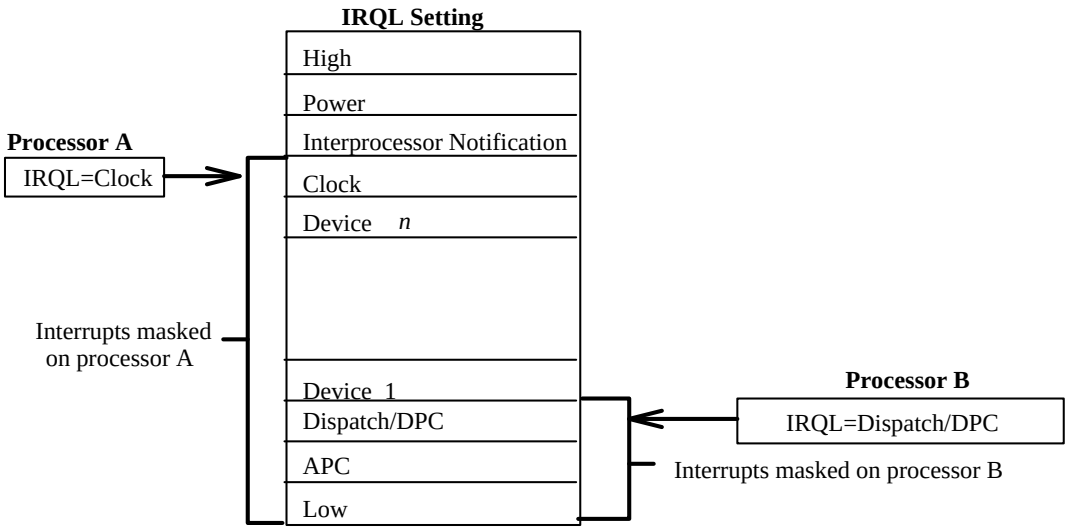


Figure 4.2. Masking Interrupts

- Mode bit (user or kernel)
- IRQL
- Temporary registers
- Stack pointer register
- Global pointer register which points to the process image
- Argument registers
- The register used to return a value to the caller

All exceptions, except those simple enough to be resolved by the trap handler, are subsequently serviced by a Microkernel routine called the *exception dispatcher*. The exception dispatcher is processor architecture-specific. The purpose of the exception dispatcher is to locate a handler that can resolve the exception. The following architecture-independent exceptions are defined by the Microkernel: memory access violations, integer overflow, floating point divide by zero, debugger breakpoint, illegal instruction, debugger single step, page read error, integer divide by zero, floating point overflow/underflows, floating point reserved operand, data type misalignment, privileged instruction, guard page violation, paging file quota exceeded.

System service calls, another type of exception, are processor architecture-specific. In the $x86$ architecture, the vector table indicates the address of the *system service dispatcher*, a Microkernel routine. The system service dispatcher looks up the system service call number in a table, captures the user-mode stack information (which contains the parameters to the call), and calls the appropriate system service routine as a procedure with the arguments captured from the user-mode stack. In the ALPHA architecture, when a *callsys* instruction is executed in user mode, the Microkernel dispatches directly to the location within the Microkernel that handles system service calls and looks in the entSys register to determine which system call is being requested. In both architectures, system service call parameters are captured and probed. The capture involves copying the arguments from user-mode memory to kernel-mode memory. The probe involves ensuring that the parameters really came from the user's address space and are valid in terms of access requirements.

4.1.2 Object Manager

The Object Manager plays a central role in the management and protection of all system resources. The Object Manager implements three principal functions:

1. Executive Object Type Management Services. The Object Manager provides type management functions that all other executive subsystems use for exporting and protecting their objects.
2. Global Name Space Services and Related Object Types. The global name space provides a means for sharing executive objects between processes via global names. To implement this name space, the Object Manager creates and exports two object types of its own: *object directories* and *symbolic links*.
3. Handle Tables and Security. User-mode processes access executive objects via a *handle*. Possession of a handle for an object implies some ability to access that object. Handles are created and managed in *handle tables* by the Object Manager. The Object Manager provides object security by calling the Security Reference Monitor (see Section "Security Reference Monitor," on page 69) before creating new handles for an object.

The remainder of this section discusses each of these functions. Additional details on specific object types can be found in Section "Objects," on page 140.

4.1.2.1 Executive Object Type Management Services

The Object Manager provides a set of generic services that all executive subsystems use for protecting and exporting their objects. Type-specific semantics are implemented by the exporting subsystem.[5]

4.1.2.1.1 Object Structure Within the executive, every instance of an object type is represented by an object structure that has two principal parts: the *object header* and the *object body*. The format of the object header is the same regardless of the object type. The Object Manager is responsible for the object header, and uses it to store information necessary for providing common object services. Some of the key information managed via the object header includes:

- Object name (optional).
- If named, a pointer to the parent directory.
- Object type identifier and a pointer to the *object type structure* (see below).
- A pointer to a *security descriptor*, if any (see Section "Security Descriptor Data Structure," on page 75).
- If opened for exclusive access, a pointer to the process that has the object opened.
- An access mode field indicating the processor execution mode (user or kernel) in which the object is accessible. Kernel-mode objects are not accessible by user-mode threads (e.g., the `type` object type described below is only available to other kernel-mode subsystems).
- A field indicating whether the object is permanent or temporary. A temporary object, which can have a name, is automatically deleted when the last handle for the object is closed.

[5] The term *exporting subsystem* is used in this report to refer to the executive subsystem responsible for managing and exporting a given object type. For example, the Process Manager is the exporting subsystem for process and thread object types (see Section "Process Manager," on page 81). Likewise, the Object Manager is the exporting subsystem for object directory and symbolic link object types.

- A pointer to the object body.

The object body is managed by the exporting subsystem for the object type. The Object Manager does not interpret or manipulate the object body (except for the Object Manager's own object types).

4.1.2.1.2 Object Types The Object Manager provides a type of object that is only available within the executive: the `type` object. Instances of a `type` object are used by executive subsystems to define other object types. For all objects, the object header points to an instance of a `type` object. For example, a process object's header would have a pointer to the `process_type` object (and the `process_type` object's header would likewise have a pointer to the `type` object).

An instance of a `type` object is represented by an object structure where the object body is managed by the Object Manager. A `type` object body includes the following information:

- A valid access mask indicating which of the available type-specific access rights are valid for this object type. Section "Overview of Object Access Rights," on page 142 provides an overview of the different types of access rights that can be associated with an object type.
- A mapping between the four generic access rights (generic_read, generic_write, generic_execute, and generic_all) and the type-specific access rights. Section "Object Type-Specific Security Attributes," on page 143 describes the type-specific access rights, and their mapping to the four generic access rights, for each executive object type.
- A mask indicating which *object attributes* are invalid for this object type. Object attributes are specified when an object is created or when an object handle is created (e.g., when a object is opened). The following attributes are possible:
 - *Inherit:* Allows an object handle to be inherited during process creation.[6]
 - *Exclusive:* Indicates that no other process may open (or inherit) a handle to the object.
 - *Permanent:* When creating an object, specifies whether an object is permanent or temporary.
 - *Case Insensitive:* Specifies whether case should be considered during name lookup.
 - *OpenIf:* When attempting to create an object, allows opening a named object if it already exist, otherwise a new object with the specified name will be created.
- Pointers to type-specific *methods*. Methods are kernel-mode procedures, provided by the exporting subsystem of an object type, that the Object Manager invokes when a service for that object type is requested by a client. These methods provide type-specific semantics to compliment the standard services provided by the Object Manager. The following type-specific methods can be defined for a given object type:
 - *Open:* Called when a handle for this type is created (e.g., an explicit *NtOpen* system call, object inheritance).
 - *Close:* Called when a handle for this type is destroyed (e.g., an explicit *NtClose* system call, process termination).

[6] Object inheritance is described later in this section.

- *Delete:* Called before the Object Manager deallocates its data structures for a given object instance.
- *Parse:* Used by executive subsystems which maintain their own name space for which the Object Manager cannot provide name parsing. This method is used by the I/O subsystem, Configuration Manager, and the Object Manager. Parse methods are fully explained below.
- *Security:* A procedure called when the security descriptor[7] for an object is set or queried. If this method is not defined, the standard security procedure is used. The I/O subsystem is the only subsystem to define a security method for its exported object type (`file_type`). The `file_type` represents several "sub-types" of objects that are exported by various file systems and device drivers.

4.1.2.2 Global Name Space and Related Objects

The Windows NT *global name space* is a hierarchical structure with a single root object directory named \ (backslash). During system initialization, the Object Manager creates the root directory. The root directory is global and is the starting point for parsing any fully qualified object pathname (i.e., all object pathnames that begin with a backslash are relative to the root directory). Additional backslashes in an object pathname serve as separators between pathname components.

An instance of any executive object type can be named, which allows the object to be shared between processes using the Object Manager's global name space (e.g., by explicitly opening the object). In general, unnamed object instances can only be shared between processes via inheritance.[8]

4.1.2.2.1 Object Manager Exported Objects The Object Manager exports two object types that implement the global object name space: object directories and symbolic links.

Object directories contain zero or more pointers to instances of other object types. Directories (and hence the object name space) may reference object types that are not exported outside the executive and are therefore not accessible by user-mode threads (e.g., instances of the Object Manager's `type` object). Windows NT provides for the hierarchical object name space by allowing directories to point to other directories.

Symbolic links, which allow for name aliasing of objects, are similar to the symbolic links in many UNIX™ systems. Symbolic links are one of the executive object types that provide a parse method. Parse methods are optionally defined for each object type by the exporting subsystem. If, when parsing a pathname, the Object Manager discovers an object other than an object directory, and there are still components left in the pathname, it attempts to invoke the parse method for that object type. If no parse method is defined for the object type, then an error is returned.

A parse method can return one of three values to the Object Manager:

- Success: name parsing is successfully completed.
- Error: the error code is returned to the caller.
- Reparse: the parse method returns a new pathname on which the Object Manager restarts pathname parsing.

[7] A security descriptor contains all the security-relevant information associated with an object. See Table 4.5 for a description of the contents of a security descriptor.

[8] Exceptions to this rule are discussed in Section "Handles and Handle Tables," on page 61.

The parse method for symbolic links takes the remaining pathname (i.e., the part of the pathname not yet parsed by the Object Manager), concatenates it with the pathname stored in the symbolic link object, and returns the new pathname on which the Object Manager restarts pathname parsing.

4.1.2.2.2 Secondary Name Spaces The I/O Subsystem (for file systems and devices) and the Configuration Manager (for the registry) implement their own hierarchical name spaces for their objects. These secondary name spaces are not implemented by the Object Manager and, except for the root of these secondary name spaces, are not part of the global object name space.

For example, the Windows NT File System (NTFS, a part of the I/O Subsystem, see Section "Windows NT File System (NTFS)," on page 96) implements a hierarchical file system on disk volumes that includes files and directories objects. From the Object Manager's perspective, the only object in the global name space is a `file_type` object instance representing the disk partition. The `file_type` object structure defines a parse method that allows NTFS to continue the name parsing and implement its own hierarchical secondary name space. While the Object Manager does not implement this secondary name space, from the perspective of user-mode processes, there appears to be a single, cohesive object name space. The Configuration Manager implements the registry name space in a similar manner.

Although the I/O Subsystem and Configuration Manager name spaces are not implemented by the Object Manager, the objects that are eventually referenced via these secondary name spaces are known since all objects, including those whose name lookup is completed by a subsystem other than the Object Manager, are still represented by an Object Manager object structure and type structure. In this way, the generic aspects of object management (including access control and protection) remains consistent for all executive object types.

4.1.2.2.3 Name Space Initialization The object name space is completely transient. It is initialized during system startup, and lost when the system is shutdown. The object name space is not permanently stored on disk. Rather, it is reconstructed every time the system is started. After the Object Manager completes its initialization, the object name space have the following structure:

Pathname	Object Type	Notes
\	directory	Root directory
`\ObjectTypes`	directory	Directory where all object types are kept
`\ObjectTypes\Type`	type	type object type
`\ObjectTypes\Directory`	type	directory object type
`\ObjectTypes\SymbolicLink`	type	symbolic link object type

Other executive subsystems will add additional object types, directories, and other objects to the object name space during system initialization.

4.1.2.3 Handles and Handle Tables

The only operations that can be performed using only an object's name are creating and opening objects, both of which result in a *handle* being created. All other actions (including object deletion) is predicated on possession of a valid handle to the object.

Handles are a 32-bit index into an executive data structure called an *handle table*.[9] A handle table describes what objects are currently accessible by all threads within a process. There is one handle table for each process. A process' handle table is kept in kernel-mode memory and hence is protected from direct user-mode manipulation. A given handle only has meaning within the context of the process for which it was created.

Handle table entries[10] contain the following information:

- *Free bit:* Indicates whether this entry is currently being used. Unused entries are linked together in a free list.
- *Inheritance bit:* Indicates that this handle should be inherited by a newly created process (see below).
- *Audit on close bit:* The audit mechanism allows object close (i.e., handle destroy) events to be audited. This bit indicates whether the Security Reference Monitor (SRM) should generate an audit event when this handle is destroyed. The current audit policy as configured by the system administrator (see Section "Audit Mechanism," on page 158) determines whether this bit is set. This feature is a efficiency measure that avoids re-checking the audit policy configuration to determine whether an audit event must be recorded when an object is closed (by the SRM making that determination and telling to Object Manager to set this bit when the handle is created).
- *Object header pointer:* Points to the header of the object structure for the object referenced by this handle.
- *Granted Access Mask:* This field indicates what access rights are permitted using this handle. The access mask can contain generic access modes (i.e., read, write, execute, all) or type-specific access modes. See Section "Access Mask Data Structure," on page 73 for more detail about access masks and Section "Objects," on page 140 for a detailed discussion of generic and specific object access rights.

User-mode processes can obtained a handle in one of five ways:

1. *Object Open:* The calling thread supplies an object pathname and requested access as arguments to an *NtOpen* system call. If name lookup and security checks are successful, then the Object Manager creates a handle for the object with the requested access rights, and places it in the handle table of the calling thread's process.
2. *Object Create:* In general, successfully creating an object can result in the creation of a handle to the object.
3. *Inheritance:* As part of process creation, the new process can be "born" with some handles that came from its parent process. The *inheritance bit* (described above) for each entry in the creating process' handle table determines whether the handle is inherited by the new process.
4. *Duplicate Handle:* If a calling process has a handle with Dup_Handle access to two other processes (the source process and the target process, either or both of which may be the same as the calling process), or has the Debug privilege, then the calling process can duplicate a handle in the source process' handle table into the target process' handle table using the *NtDuplicateObject* system call.

[9] The handle table is also called an *object table* in much of the Windows NT documentation. Throughout this report, the term handle table will be used exclusively.

[10] While a handle is technically just a 32-bit index into the handle table, the term is also commonly used to mean the handle table entry to which it refers.

If the access rights requested for the new handle are equal to or a proper subset of the source handle's granted access mask, then handle duplication will occur without further access validation. However, if the requested access rights are not a subset of the source handle's granted access mask, then the Object Manager forces full access validation for the requested access rights (i.e., as if this were an *NtOpen* call). In this case, access validation will use the security context of the calling process (and not the source process) to validate the requested access rights. In either case, if successful, the *NtDuplicateObject* call will result in a new handle in the target process (and no changes to the source and calling processes).

5. *Special Cases:* There are special interfaces that allow a handle to be created for three of the executive object types. The executive system calls *NtCurrentProcess* and *NtCurrentThread* create a handle for the process and thread of the calling thread. Likewise, the system calls *NtOpenProcessToken* and *NtOpenThreadToken* create a handle for the token object (See Section "Tokens," on page 70) associated with a process or a thread.

4.1.3 Memory Manager

This section describes key aspects of virtual and physical memory management, and summarizes the key Windows NT Memory Manager differences between the two supported processors: the Intel $x86$ and the Digital Alpha DECchip 21064-AA.

4.1.3.1 Virtual Memory Management

A virtual address space in Windows NT is 2^{32} or 4 GB long. Every process has one associated virtual address space. The high-order 2 GB of a virtual address space is reserved for the executive and is not accessible by user-mode threads. Each address space is unique for each process (i.e., distinct mapping tables), though there may be some shared physical pages as discussed below (e.g., much of the kernel-mode half of an address space is shared by all address spaces).

The separation of a virtual address space into user-mode and kernel-mode halves is based on the hardware-enforced restrictions in the MIPS R4000 processor.[11] The Alpha Windows NT PALcode implements this restriction transparently to the Memory manager, i.e., it appears as a hardware restriction (see Section "Translation Buffers," on page 28). For Intel $x86$ processors, this restriction is enforced by the way page tables and page directories are constructed.

4.1.3.1.1 Address Translation Figure 4.3 illustrates virtual-to-real address translation for Windows NT (using a 4 KB page size). Virtual addresses are translated to real addresses via a set of process-unique page directories and tables. Windows NT's paging mechanism is based on the paging hardware provided by the Intel $x86$ processors (see Section "Paging," on page 21). A paging mechanism very similar to that of the $x86$ processor is implemented by the Alpha Windows NT PALcode (see Section "Windows NT Virtual Address Translation," on page 32).

The page size is 4 KB for the $x86$ processor and 8 KB for the Alpha processor. For both processors, a 32-bit virtual address has three fields: directory offset, page table offset, and page offset. The directory offset selects a page directory entry from the process' page directory. The base address of the current process' page

[11] The MIPS R4000 processor is not included in the evaluated configuration. However, the R4000 is supported by Windows NT and much of the Memory Manager design is influenced by the R4000 memory architecture.

directory is kept in the CR3 register for Intel $x86$ processors. For the Alpha, the PALcode-defined register PPDR points to the current process' page directory. The PALcode maintains one PPDR register per process and swaps the current PPDR as part of a context switch.

A Page Directory Entry (PDE) contains an address to a page table. The page table offset field of a virtual address selects a *Page Table Entry (PTE)* from this page table. For non-shared memory, this PTE contains a physical page frame address. The page offset field indexes into the selected physical page frame.

For a shared memory page, the Memory Manager uses a *prototype PTE* mechanism to efficiently manage invalid PTEs.[12] When an invalid PTE representing a shared page is discovered by the Memory Manager, the contents of the PTE identify a prototype PTE. The prototype PTE contains the physical page frame (if in memory) or disk location (if not in memory) of the referenced page. The memory manager uses this information to construct a valid PTE for the page table. All processes that share this page use this prototype PTE in a similar manner.

In the Alpha, during virtual address translation, the hardware determines when a TLB miss occurs and raises a hardware exception. The Alpha PALcode handles this exception and performs the page directory and table translation. If the requested page is in memory, the PALcode refills the TLB. For the $x86$ processor, the hardware handles TLB misses, page directory and table translations, and TLB refills. For both processors, the Memory Manager only becomes involved in virtual address translation when a page fault occurs (e.g., the page is not in memory or is invalid).

4.1.3.1.2 Section Objects and Shared Memory Processes can have two types of memory: *private* and *shared*. Private memory is not shared between processes[13] and is backed by the system paging file.

Shared memory provides a means for two processes to communicate via pages of physical memory common to both virtual address spaces. Shared memory is provided via an Windows NT object type called *sections*. Sections are like any other object in Windows NT. They have an associated ACL, can be named, and are accessed via an opened handle.

Sections can be backed by the system paging file or by a file (i.e., memory mapped file). When backed by a file, a section cannot be created unless the creator has an opened handle to the file. Once created, the section object (and hence the backing file) can be accessed by opening the section (the file need not be opened). However, the section object cannot be opened for greater access than the creator had to the original file, regardless of the access permitted on the section's DACL. For example, for a section backed by a file for which the creating process had read-only access, the section may only be used to map read-only views to the file, even if the section's DACL allows write access.

Once opened, a thread may map *views* of the section into its process address space. Multiple views may be mapped to the same opened section object, each of which corresponds to all or part of the section object.

[12] An invalid PTE is one that cannot be automatically translated by the hardware for the $x86$ processor and by the PALcode for the Alpha processor. Typically, an invalid PTE represents a page that is not currently in memory and must be moved into memory by software. Invalid PTEs will result in an exception that the Memory Manager handles.

[13] Private memory is logically private, but Windows NT may use copy-on-write protected pages for private memory. This in effect uses the same physical pages of memory for private memory in different processes when only read access is attempted. However, processes are unaware of this sharing. See Section "Page Protection," on page 66 for more information about copy-on-write protection.

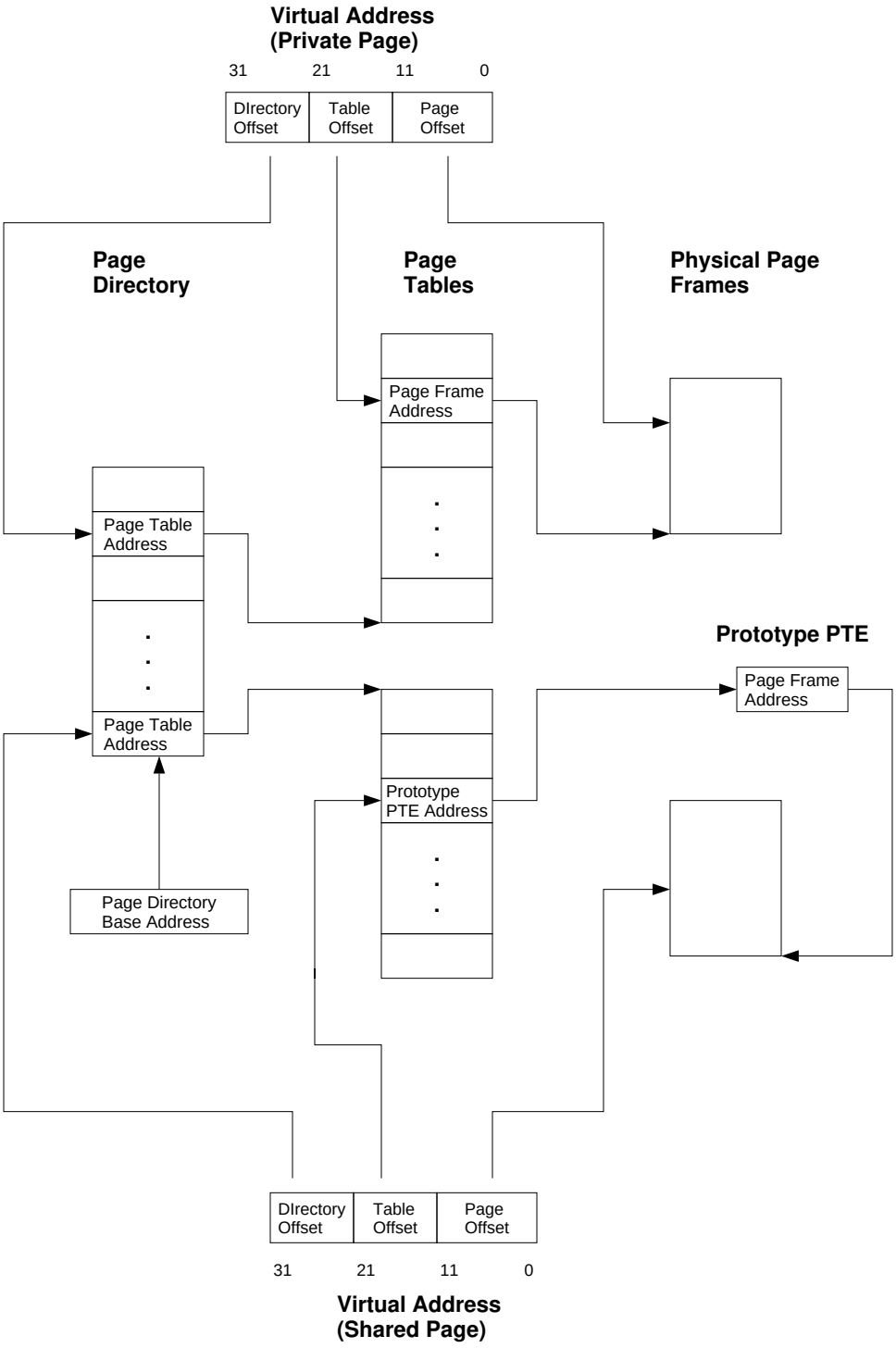


Figure 4.3. Windows NT Virtual Address Translation (4 KB Page Size)

4.1.3.1.3 Memory Allocation A memory *region*, which is a range of a virtual addresses within a virtual address space, may be in one of three different states: *free, reserved,* or *committed*. All of the address space is initially free, and remains that way until reserved or committed. Reserved regions are allocated, but not associated with backing store. Committed regions are backed by the paging file or (optionally for sections) by a mapped file. Only committed memory regions are accessible. Both free and reserved addresses are inaccessible.

4.1.3.1.4 Page Protection Committed memory regions can have the following types of protection available:

No-access: Any attempt to access this page results in an access violation exception.

Read-only: Write access is not permitted.

Execute-only: Instructions may be fetched from the page. Neither the Intel $x86$ nor the Alpha processor support execute-only access on pages. Thus, execute-only is equivalent to read-only for both of these processors.

Read-write: Both read (execute) and write access is permitted to the page.

Copy-on-write: This type of access is only meaningful when sharing physical memory between multiple processes. Read and write access is permitted to the page; however, any attempt to write the page causes a private copy of the affected page to be made for the modifying process. As long as only read accesses are attempted, the PTE will point to a prototype PTE which may be shared with other processes. This access type allows multiple processes to share physical pages for logically private memory regions, even if write access is permitted.

Execute-read: For the $x86$ and the Alpha processors, this is equivalent to read-only access.

Execute-read-write: For the $x86$ and the Alpha processors, this is equivalent to Read-Write access.

Execute-copy-on-write: For the $x86$ and the Alpha processors, this is equivalent to copy-on-write access.

Guard page: Guard pages support the ability to automatically check boundaries of data structures such as stacks, and to dynamically extend these structures. Guard pages have some form of read-execute-write access protection in addition to being identified as a guard page (no-access protected pages cannot be guard pages). When an access attempt is made to a guard page, the Memory Manager raises a guard-page exception which the calling thread can handle in an appropriate manner (e.g., cause an overflow error, dynamically increase the data structure).

4.1.3.1.5 Virtual Address Descriptors When a thread allocates or commits a free region of memory, Windows NT does not immediately create the necessary page tables and PTEs. Rather it waits until a page is addressed before creating these structures. To keep track of memory regions which have been allocated or committed, but for which PTEs do not yet exist, the Memory Manager associates a set of *Virtual Address Descriptors (VADs)* with each process. Each VAD contains the range of addresses it describes, page protection, inheritance attributes (see below), type of memory (i.e., private or shared), and for mapped views of sections (i.e., shared memory), information describing the associated section object. On the first attempt to access a page in a new region (after the region has been committed), the appropriate page table and PTEs are created using the descriptive information from the VAD.

The *inheritance attribute* is used when creating a new process. The *NtCreateProcess()* executive system call will, by default, define the address space of the new process based on the inheritance attributes in the VAD for the parent process.[14] Memory regions whose VADs indicate inheritance will become part of the new process' allocated memory regions. For committed pages in inherited regions that are private, Windows NT uses copy-on-write protected pages rather than physically copying the pages when creating the new process.

4.1.3.2 Physical Memory Management

The Memory Manager maintains a *Page Frame Number (PFN)* database to help manage physical memory. This database holds state and usage information for each physical page of memory. The memory manager uses this information for making memory allocation decisions. Each physical page will always be in one of six possible states:

- *Valid:* At least one valid PTE or prototype PTE refers to this page (i.e., the page is currently in use).
- *Standby:* Each process is allowed a certain number of pages of memory, called its *working set*,[15] which limits the amount of physical memory any one process can control. When a process' working set is full, and a new page must be paged in from the paging file or mapped file, the Memory Manager selects a page frame to remove from the process' current working set in order to make room for the new page. A standby page frame is one that is no longer in any process' working set (i.e., there is no valid PTE for it) but has not been freed. A standby page frame can be put back into the process' working set without the need for disk I/O.
- *Modified:* A modified page frame is similar to a standby page frame except that the page has been modified and needs to be written to disk before being reused for a different virtual page.
- *Zeroed:* The page is unused and initialized to all zeros.
- *Free:* The page is unused but has not been initialized to zeros.
- *Bad:* The physical page frame has a hardware error and is unusable.

Using this state information and a set of pointers associated with each PFN database entry, the Memory Manager maintains five lists of unallocated memory:

- Zeroed list,
- Free list,
- Standby list,
- Modified list, and
- Bad list.

The free list and zeroed list are used first when allocating new page frames to a virtual page. If an uninitialized page is adequate (e.g., the page frame will be completely overwritten from a mapped file before being made accessible to a thread), then the Memory Manager attempts to allocate a page from the free list. If an

[14] The parent process is not necessarily the process of the thread calling the *NtCreateProcess* system call. Rather, the calling thread provides a parent process handle as an argument to the *NtCreateProcess* system call.

[15] The size of a process' working set varies according to system-specific resource control policies and the demands for memory.

initialized page is necessary (e.g., a new private page is needed), then the Memory Manager attempts to allocate the page from the zeroed list. If the zeroed list is empty, then the Memory Manager will try to allocate a page from the free list, zero it, and then make it accessible. If the free list is empty, the Memory Manager will then try to allocate first from the standby list, and then from the modified list. In either case, the allocated page frame will be zeroed or overwritten before being released to the process. Page frames are not allocated from the bad list.

The Memory Manager uses several kernel-mode threads to help manage these lists. The *zero page thread* periodically takes page frames from the free list, zeros them, and puts them in the zero list. The *modified page thread* periodically takes page frames from the modified list, writes them back out to their backing file (i.e., the paging file or a mapped file), and moves the page frames to the standby list.

Using the PFN database, the Memory Manager ensures that whenever a page frame is allocated or reallocated to a new virtual page, its contents are either completely overwritten with information from a backing file, or its contents are zeroed before the page is made accessible to any process.

4.1.3.3 Summary of Key Processor Differences

This section summarizes some of the key Memory Manager architectural differences as a result of Windows NT's support for both the Intel $x86$ and Digital Alpha processors. Some of these differences have already been described. Early in this report it was noted the separation of the virtual address space into kernel-mode and user-mode halves is implemented differently on each processor. For the $x86$ processor, the Memory Manager simulates this separation by the way it constructs page tables for each address space. For the Alpha processor, this separation is enforced by the Windows NT PALcode and appears to the Memory Manager as a hardware restriction.

Section "Address Translation," on page 63 notes that the paging mechanism is implemented in hardware for the $x86$ processor and by PALcode for the Alpha processor. Also, the page size is 4 KB in the $x86$ processor and 8 KB in the Alpha processor.

Additional architectural differences are described below.

4.1.3.3.1 TLB Flushing on Context Switches On the Intel $x86$ the processor itself manages the TLB. The $x86$ processor automatically flushes the TLB on a context switch. In the Alpha, the PALcode performs most of the TLB management, but with some differences from the way the $x86$ processor manages its TLB.

An entry in the Alpha's TLB can represent a "global" page (such as the kernel-mode executive code which is present in all processes). Since such pages are a part of every address space, they need not be flushed from the TLB as part of a context switch. The Alpha PTE format includes a global bit (see Section "Translation Buffers," on page 28) which when set by the Memory Manager, will tell the PALcode to set the corresponding TLB entry ASM bit (which indicates a global page) when the page is loaded in the TLB. TLB entries with the ASM bit set will not be flushed during a context switch (though these "global" pages may be replaced as part of the normal paging mechanism).

Another feature for controlling TLB flushing is the Address Space Number (ASN). In the R4000 Windows NT executive, every process is assigned a unique ASN, which is loaded into the R4000 TLB whenever a TLB entry is refilled. The R4000 address translation mechanism checks for matches between the ASN of a TLB

entry and the ASN of the current process, thereby preventing the possibility of an address lookup for one process matching an identical virtual address for another process in the TLB. Because ASNs are unique, the TLB need not be flushed during a context switch. The Alpha architecture definition supports an ASN construct nearly identical to that of the R4000, and the PALcode exports this construct to the Windows NT Executive. However, in the DECchip 21064-AA implementation of the Alpha architecture, ASNs are *not* supported within the chip's TLBs. Thus, while Windows NT sets and uses the ASN for memory and processes, they are not actually used to prevent TLB flushing by the Alpha PALcode. The Alpha PALcode currently flushes all TLB entries (except those marked global) as part of a context switch.

4.1.3.3.2 Write Protection A page of memory in Windows NT can be protected against write access. Write protection is implemented via the hardware paging mechanism for the Intel $x86$ processor, which provides a write protection bit for each page table entry (see Section "Windows NT Virtual Address Translation," on page 32).

In the Alpha processor, paging mechanisms are implemented in PALcode. An Alpha PTE does not provide a write protection bit. Instead it provides a "dirty" (D) bit. The D bit serves two purposes: it provides write protection and it indicates that a page has been modified. For the Alpha, all PTEs initially have the D bit cleared. If a process attempts to write a page with the D bit cleared, the PALcode will raise an access violation exception. The Windows NT exception handler will examine the access mode for that page (by examining the software maintained entry in the page tables) and determines whether that page is writable. If the page is writable, the D bit is set and the address translation restarted. Subsequent write attempts to that page will not generate an exception (since the D bit is now set). Otherwise, the write attempt will fail.

4.1.3.3.3 Large Page Sizes for the Alpha Page sizes are fixed at 4 KB for the Intel $x86$ processor. The Alpha processor allows the page size to range from 8 KB up to 4 MB. For the Alpha, Windows NT allows for special cases where the page size is larger than 8 KB. This special case is only used by the kernel-mode video driver, allowing it to map all of video RAM using one TLB entry. For such special cases, the "granularity hint" field in the Alpha PTE entry will specify the page size (see Section "Translation Buffers," on page 28).

4.1.4 Security Reference Monitor

The Security Reference Monitor (SRM) is an executive subsystem that provides functions to support security policy enforcement. These functions are provided to the protected servers as well as the other executive subsystems. The major functions of the Security Reference Monitor (SRM) are listed below.

- Access validation
- Privilege testing
- Audit generation

The SRM's executive access validation functions are mostly used by the Object Manager. Before the Object Manager releases a handle to an object, it calls the SRM's access validation services to determine if the desired access should be granted. The privilege testing services are used by each executive subsystem whose services require privilege. Privileges are described in Section "Privileges," on page 151. Executive subsystems also use the SRM audit functions to audit access to their objects and the use of privilege.

The SRM's user-mode functions allow protected servers to protect their own exported objects. Protected Servers that export their own objects call the SRM's access validation, audit generation, and privilege testing functions. The audit functions require the protected server to have the Audit privilege.

The token object is the basis for these three major functions and is exported by the SRM. The token object, which is described below, is used to store security related information for a process.

Servers use the concept of *impersonation* to identify clients to the SRM on whose behalf the protected server is making a request. Impersonation allows one process to act on behalf of another, taking on the security attributes of that process.

Another function of the SRM is to track logon sessions. Specifically, the SRM tracks tokens associated with a logon session.

4.1.4.1 Tokens

A token object contains the security attributes of a process or thread and is used to perform access validation. Each process has a token, called a *primary token*. A thread may also have a token, called an *impersonation token*. A *security context* refers to the security related information associated with a process (the information in the process' token). The security attributes of a process which are stored in the token is described in Table 4.3. Note that the Security Identifier (SID) in the token is used to uniquely identify either a user or a group. For further information about the SID see Section "Identification and Authentication," on page 181.

4.1.4.1.1 Token Services The SRM provides two ways to create tokens: creation and duplication. While the result of each one is a token, only the create service actually creates a new token defined by input parameters. Duplication makes a token by copying a subset of the contents of an existing token. The create service requires a privilege (CreateToken) and is used exclusively by the LSA.

Other token functions provided by the SRM include: opening, reading certain field values, changing certain field values, enabling and disabling privileges, and enabling and disabling groups. Some functions require the thread to have a valid handle to the token. The function provided to change token fields requires that the token be opened with TokenAdjustDefault access. The fields that can be changed with these services are the owner and default Access Control List (ACL) fields. The functions that enable and disable privileges require TokenAdjust access. These functions can only set the privileges that are assigned in the token to be enabled or disabled, they cannot assign new privileges to the token or eliminate privileges from the token. The functions that enable and disable groups requires that the token be opened with TokenAdjustGroups access and can only set groups already defined in the token to be enabled or disabled. These functions cannot add new groups to the token or take existing groups out of the token. The duplication service requires that the token be opened with TokenDuplicate access.

A token operation provided by the SRM only to other executive subsystems is creating a SYSTEM token. A SYSTEM token defines the SYSTEM security context as described in Table 4.9, on page 107.

The SRM also supports operations which do not manipulate the token but are token related. They are: inquiring if a thread is impersonating, assigning a token for impersonation, discontinuing impersonation, and assigning a primary token to a process. Assigning a primary token to a process requires the AssignPrimaryToken privilege.

Token Field	Description
User Security Identifier (SID)	Uniquely identifies the authenticated user on whose behalf this token was created.
Group SID(s)	Group SID(s) the user is a member of and group attributes (Enabled, Disabled).
Privilege(s)	Privilege(s) assigned to the token and privilege attributes (Enabled, Disabled).
Owner	Points to an SID that will be assigned as the owner of any objects created on behalf of the user represented by the token. This SID must be one of the user or group SIDs already in the token.
Default DACL	Points to an ACL that will be assigned by default to any objects created by the User SID.
Type	Identifies if the token is primary or impersonation.
Impersonation Level	Valid only if token type of impersonation type. Indicates the level of impersonation (Impersonate, Identify, Anonymous).
AuthenticationID	Uniquely identifies the logon session this token represents.
Statistics: - TokenID - Dynamic Charged - Dynamic Available - Group Count - Privilege Count - Modified ID	 Uniquely identifies this instance of token object. Memory amount allocated, in bytes, for storing default protection and a primary group ID. Memory amount allocated for storing default protection and a primary group ID that is not already in use. Number of groups SIDs in token. Number of privileges in token. An identifier that changes each time the token is modified. Used in quick tests to see if the security context has changed since last used.

Table 4.3. Token Data Structure

4.1.4.1.2 Impersonation Windows NT allows a process to take on the security attributes of another process through a technique called impersonation. This capability allows threads to have a different set of security attributes than that of its process. This thread is then said to be impersonating the other process.

Each process has one primary token which is used for access validation, unless the thread in this process requesting access has an impersonation token. An impersonation token is associated with a thread and it is optional. It represents the security attributes of another process and, if present, supersedes the primary token of the thread's process for access validation. The type field in the token object indicates whether a token is a primary or an impersonation token. The Impersonation Level field is only valid for impersonation type tokens and is defined by the thread that is being impersonated.

There are a number of security parameters that threads may specify to control how it may be impersonated. These parameters are referred to as Security Quality of Service (SQOS) parameters. This can best be described by using the client-server model which uses impersonation to allow a server thread to impersonate a client thread. When clients connect to servers to request services, they specify the SQOS parameters at this time which controls the degree of impersonation (see Section "LPC Impersonation," on page 86). The services the client requests of the server must be consistent with these parameters. The SQOS parameters are listed below.

- Impersonation Level: indicates the degree to which the server can impersonate the client. The client must specify a high enough impersonation level to allow the server to perform the services it requests.
 - Impersonation - allows the server to act as the client to accomplish the requested service (e.g., opening a file on the client's behalf).
 - Identification - allows the server to know who the client is (IDs, privileges) without being able to impersonate the client. The server can perform services that only requires the client's identification (e.g., make access validation decisions).
 - Anonymous - indicates the server will not know the client's identity, providing no capability to the server. An anonymous token is supported for symmetry of implementation purposes (e.g., so that an API that supports both identified users and anonymous users need not have special parameters depending on the type of services—the client would just pass either an identification or an anonymous token depending on the type of service desired).
- SecurityContextTracking: a boolean flag indicating whether the server's copy of the client's token is constant (static tracking) or kept consistent (dynamic tracking) with the client's token (i.e., by value or by reference).
- Effective Only: a boolean flag indicating whether the entire token is made available to the server or only the enabled aspects (privileges and groups). If the entire token is available, the server may enable and disable privileges and groups held by the client that were disabled at connection time.

4.1.4.2 Access Validation

This section describes the access validation routine and the data structures used to support validation.

4.1.4.2.1 Access Validation Data Structures Access validation is performed by the SRM upon request from executive subsystems and servers. The Object Manager ensures this validation is requested before creating a handle to an executive object. The only executive objects for which access validation is not called directly by the Object Manager are the file, device, and key object types. If access is granted, the granted access is stored in the handle to the object. Each subsequent access to that object by that process using that handle is checked against the access granted at the time the object was opened (see Section "Handles and Handle Tables," on page 61).

For objects which are maintained by servers, the server is responsible to ensure that this validation routine is requested before it opens an object. If successful, the server stores this granted access and, like the executive, each subsequent access to that object is checked against this granted access mask.

To perform access validation the SRM uses the token of the process or thread, the desired access mask, and the security descriptor of the object. These data structures are described below.

4.1.4.2.1.1 Access Control List Data Structure There are two categories of Access Control List (ACL): the Discretionary Access Control List (DACL) and the System Access Control List (SACL). The DACL represents discretionary access control on objects and the SACL represents the auditing control on object accesses. Access to the SACL requires a privilege (SecurityPrivilege). Therefore, setting (and modifying) the control to audit access to a particular object is a privileged operation. ACLs contain an ACL

header and a list of Access Control Entrys (ACEs). The ACE format and content is listed in Table 4.4. The ACL header contains information such as the size of the ACL and the number of ACEs stored in the ACL. There are two types of DACL ACEs and one type of SACL ACE:

- DACL ACE Types
 - AccessAllowed - used to grant access to a user or group of users
 - AccessDenied - used to explicitly deny access to a user or group of users.
- SACL ACE Type
 - SystemAudit - used to control audit generation of access.

4.1.4.2.1.2 Access Mask Data Structure The access mask is a data structure used to describe access rights. It is used within the ACL structure for objects and to specify a desired access in an access request. Each bit in the mask represents a particular access rights as shown in Figure 4.4. There are three categories of access types to objects: standard, specific, and special. The standard access type applies to all objects. The specific access type are object type-specific access rights and apply only to a single object type. The definition of an object includes its type-specific accesses. The special access type are access rights which require some unique processing. The special access type includes: generic access types, maximum allowed access (MA), and system security access (AS). Generic access types are high level access rights (e.g., read, write, execute) which map to a different set of specific access rights per object type. (See Section "Overview of Object Access Rights," on page 142 for more information.) The maximum allowed access right can only be used to specify a desired access mask in an access request—it cannot be used in a DACL. It is used to request the maximum access allowed by the protection assigned to that object. The system security access provides read and write access to the SACL. The access mask data structure is shown in Figure 4.4.

The SRM provides a routine which maps (converts) generic access rights to specific access and standard access rights. This routine is called before the SRM's access validation routine is called to convert the generic access rights in both the a request access mask and in an ACL access mask.

A request access mask is used when an object is created or opened and it is these create and open routines in the Object Manager which call the SRM to convert the generic access rights. This conversion routine returns an access mask with the appropriate mapping to specific and standard access rights and it is this access mask which is sent back to the SRM to perform access validation. The access masks used to protect objects in the ACLs are mapped by the SRM conversion routine before the access mask is placed in the ACL.

The access validation routine does not consider generic access rights because the request access mask and the ACL access mask do not contain any generic access rights. Therefore, the granted access mask returned by the validation routine (if allowed) does not contain any generic access rights and it is this access mask which the Object Manager places in the handle of the object.

4.1.4.2.1.3 Security Descriptor Data Structure A security descriptor contains the security information associated with an object. The ACL described above is part of this data structure. The contents of the security descriptor is listed in Table 4.5.

4.1.4.2.2 Access Validation Algorithms When a call to the SRM is made for access validation, there are two possible algorithms used to perform this validation. One is used if the calling thread's requested

ACE Field	Description
Inheritance Control Flags	Boolean flags used to control inheritance characteristics of the ACE. • ObjectInherit - inherited by sub-objects of the container. • ContainerInherit - inherited by sub-containers of the container. • NoPropagateInherit - inheritance control flags are not propagated in ACEs inherited by sub-containers. • InheritOnly - ACE is not to be used in validating access attempts to the corresponding object, but will be applied to sub-objects as they are created.
ACEType	Type of ACE (AccessAllowed, AccessDenied, SystemAudit).
ACESize	Size of the ACE in bytes.
ACETypeSpecific	This field depends on the type of ACE. • Control Flags - boolean flags only relevant for the SACL ACE types. – SuccessfulAccess - audit messages (depending on ACEType field) should be generated for successful accesses. – FailedAccess - audit messages (depending on ACEType field) should be generated for failed accesses. • Access Mask - For each ACE type the specified access mask in this field means something different (see Section "Access Mask Data Structure," on page 73): – AccessAllowed - access mask indicating which accesses are granted to the specified SID. – AccessDenied - access mask indicating which accesses are denied to the specified SID. – SystemAudit - access mask indicating which accesses are to cause audit messages to be generated. The SuccessfulAccess and FailedAccess control flags indicate whether messages are to be generated for successful accesses, unsuccessful accesses, or both. • SID - the SID value to which the access mask field is applied.

Table 4.4. ACE Format

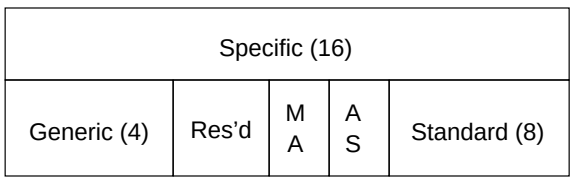


Figure 4.4. Access Mask

Security Descriptor Field	Description
Revision	Revision level, allows identification of future changes.
Control	Set of boolean flags qualifying the meaning of individual fields. It indicates what components the security descriptor includes as well as whether the included components were explicitly provided or were obtained by default. Note that implicit or explicit definition is important in assigning security to new objects. • OwnerDefault - indicates that the owner SID pointed to by the Owner field was provided by default. This may affect the treatment of the SID with respect to inheritance of an owner (ignored if Owner flag is null). • GroupDefault - indicates that the Group field was provided by default. This may affect the inheritance of the primary group (ignored if Group field is null). • DaclPresent - indicates that a discretionary ACL is present in the DACL field. If set, and the DACL field is null, then a null ACL is being specified (no access granted). Note that a null ACL is different from an absent ACL (unconditional access granted). • DaclDefaulted - indicates that the ACL pointed to by the DACL field was provided by default. This may affect the inheritance of the ACL (ignored if the DaclPresent flag is not set). • SACLPresent - indicates that a system ACL is present in the SACL field. If set, and the SACL field is null, then an empty ACL is being specified. Note that an empty ACL is different from an absent ACL. • SACLDefaulted - indicates that the ACL pointed to by the SACL field was provided by default. This may affect the inheritance of the ACL (ignored if the SACLPresent flag is not set).
Owner	Pointer to an SID representing the object's owner.
Group	Pointer to an SID representing the Object's primary group. If null, then no primary group is associated to the object.
SACL	Pointer to a system ACL (see Access Control List data structure). Only valid if the SACLPresent control flag is set. If the SACLPresent flag is set and this field is null (or zero), then a null ACL is specified.
DACL	Pointer to a discretionary ACL (see Access Control List data structure). Only valid if the DaclPresent control flag is set. If the DaclPresent flag is set and this field is null (or zero), then a null ACL (no access granted) is specified.

Table 4.5. Security Descriptor Data Structure

access (desired access mask) is the maximum access allowed access right, and the other is used if the calling thread's desired access mask is not the maximum access allowed access right. Each one is described below.

4.1.4.2.2.1 Access Validation for Maximum Allowed Access To perform this validation, the SRM uses the ACL of the target object and the token of the thread requesting access, and returns a granted access mask. The objective of the algorithm is to grant all access rights allowed by the DACL which was not explicitly denied in any **earlier** DACL entries. The algorithm accomplishes its goal by examining each ACE in the DACL and computing two access masks: a granted access mask and a denied access mask. Before any bit is added to each mask, the other mask is checked to be sure it is not contained in the other mask. Only if it is not contained in the other mask will it be added to a mask. The steps the algorithm performs is listed below.

- For each ACE which contains a Security Identifier (SID) that matches a SID in the requesting thread token, one of the following steps is performed depending on the ACE type:
 - AccessAllowed ACE type: this ACE's access mask is added to the computed granted access mask, only if it is not already contained in the computed denied access mask.
 - AccessDenied ACE type: this ACE's access mask is added to the computed denied access mask, only if it is not already contained in the computed granted access mask.
- When all the DACL entries have been examined, the computed granted access mask is returned to the caller.

4.1.4.2.2.2 General Access Validation Algorithm The objective of the access validation algorithm is to compare the desired access mask and the token of a process against the DACL of the target object. This algorithm attempts to clear all the access bits in the desired access mask by examining each ACE in the DACL. The result of the algorithm is either a failed status or a granted access mask. As stated earlier, there are three types ACEs: System, Allowed and Denied. If a match is found in a Denied ACE, the request immediately fails. If a match in an Allowed ACE then the access right in that ACE is cleared from desired access mask. This process continues until the desired access mask is totally cleared, a failure is reached, or the end of the DACL is reached. If the end of the DACL is reached and the desired access mask is not totally cleared, the requested access is denied. The SACL itself, which controls auditing of object access, is not evaluated by the access validation algorithm. However, access to the SACL is considered by the algorithm. The steps performed in the access validation routine are listed below.

1. There are two access rights that this algorithm knows about. If a certain privilege is enabled in the calling thread's token, these access rights are granted regardless of the contents of the ACL. Therefore, when these access rights are requested a privilege test is called which is described in Section "Privilege Testing," on page 77. These special access rights are: the SystemSecurity access which requires the Security privilege, and WriteOwner access which with the TakeOwnership privilege overrides the DACL. If the privilege test results are true, those access rights are cleared in the desired access mask. If the desired access mask is totally cleared then access is granted and the algorithm ends.
2. If the DaclPresent control flag in the security descriptor is not set, this implies that a DACL is not present (implying no protection of the object) and access is unconditionally granted. This will occur even if a DACL is present but the control flag is not set.[16]

[16] This will not occur for protected objects, they will have default DACLs.

3. If the requester is the owner of the object, the ReadControl and WriteDac access rights in the desired access mask are cleared. If the desired access mask is now totally cleared, access is granted.

4. Each ACE in the ACL is then evaluated in the following way, if the desired access mask is totally cleared at any time, access is granted which completes the algorithm.

 - The SID in the ACE is compared to the set of SIDs in the user's access token. If a match is not found, the ACE is skipped and processing continues with the next ACE (step 4). If found, processing continues to the following step and is based upon the type of the ACE.
 - AccessDenied ACE type: The ACE access mask is compared with the desired access mask. If there are any accesses in both masks, further processing is not necessary and access is denied. Otherwise, processing continues with the next ACE.

 Note: This algorithm is not cognizant of the ACE order. However, ordering is important as it effects the result of this algorithm.
 - AccessAllowed ACE type: The ACE access mask is compared with the desired access mask. For each match, the corresponding access is cleared in the desired access mask. If the desired access mask is totally cleared, further processing is not necessary and access is granted. Otherwise, processing continues with the next ACE.
 - For the system ACE types: The ACE is ignored and processing continues with the next ACE.

5. At the end of the ACL evaluation, if the desired access mask is not totally cleared, access is denied.

4.1.4.2.2.3 ACL Order and Access Validation Algorithms The access validation algorithms do not enforce a particular ordering of the ACEs in the ACL. The algorithm is performed upon the ACL passed into it with no modification of its order. However, the order of the ACL does impact the result of the algorithm.

The administrator must use the ACL editor provided by the File Manager (see Section "File Manager," on page 129) to generate and modify ACLs. It is important to note that this ACL editor does enforce a particular ordering of the ACL as it modifies and generates ACEs on behalf of its user. The ACL editor places all AccessDenied ACE type ACEs at the beginning of the ACL. Therefore, the algorithm will return "no access granted" if the particular access is denied to a user or group in one ACE, even though it may be granted to that user or group in another ACE. Note that all users may use this ACL editor.

4.1.4.2.3 Access Determination Example A token and DACL of an object is shown in Figure 4.5. The token represents the user FredMgr and the ACL is evaluated as follows: The first ACE is evaluated and does not grant any requested accesses. Processing continues to the next ACE where it is evaluated and the Read access is cleared from the desired access mask. Since all requested access are still not cleared, processing continues to the third ACE. This ACE is evaluated and all accesses are still not cleared form the desired access mask. Since desired access is not totally cleared and the end of the ACL has been reached, access is denied.

4.1.4.3 Privilege Testing

The SRM is responsible for determining if required privileges are enabled for the thread attempting a privileged operation. The SRM does not determine what privileges are necessary to perform each of these operations. The subsystem or server responsible for implementing that operation's functionality must send

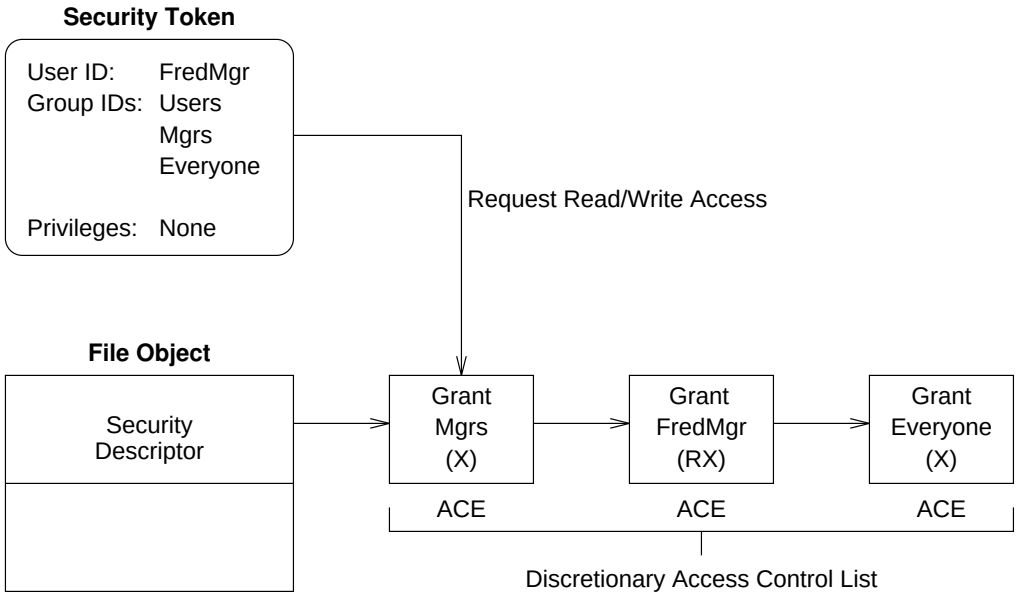


Figure 4.5. Access Determination Example

the SRM this information. The SRM is passed the required privilege set to perform the operation and the token associated with the thread. The token is then checked by the SRM to determine if the required privileges are enabled and the result is then passed back to the requesting subsystem or server. Privileges are described in Section "Privileges," on page 151 and a description of each privilege is listed in Table 5.5, on page 154. The operations that require a privilege fall into the following two categories:

- Privileged object accesses
- Privileged system services

Privileged object accesses are access rights which are either not controlled by an object's DACL or can override the DACL with a particular privilege. The system security access, which provides access to the SACL, is an access right which requires a privilege and is not controlled by the DACL. This access may only be granted to users with the Security privilege. An example of an object access that may be granted when a privilege is held and is otherwise controlled by the DACL is the WriteOwner access which allows the owner field in the security descriptor to be modified. This access may be granted to users with TakeOwnership privilege even if the DACL of an object does not allow this access. The Backup and Restore privileges also override the DACL.

Privileged system services are services in Windows NT which require a specific privilege (e.g., changing the system time). The subsystem or server responsible for implementing these services are responsible for calling the SRM to test the process' associated token to determine if a specific privilege is present.

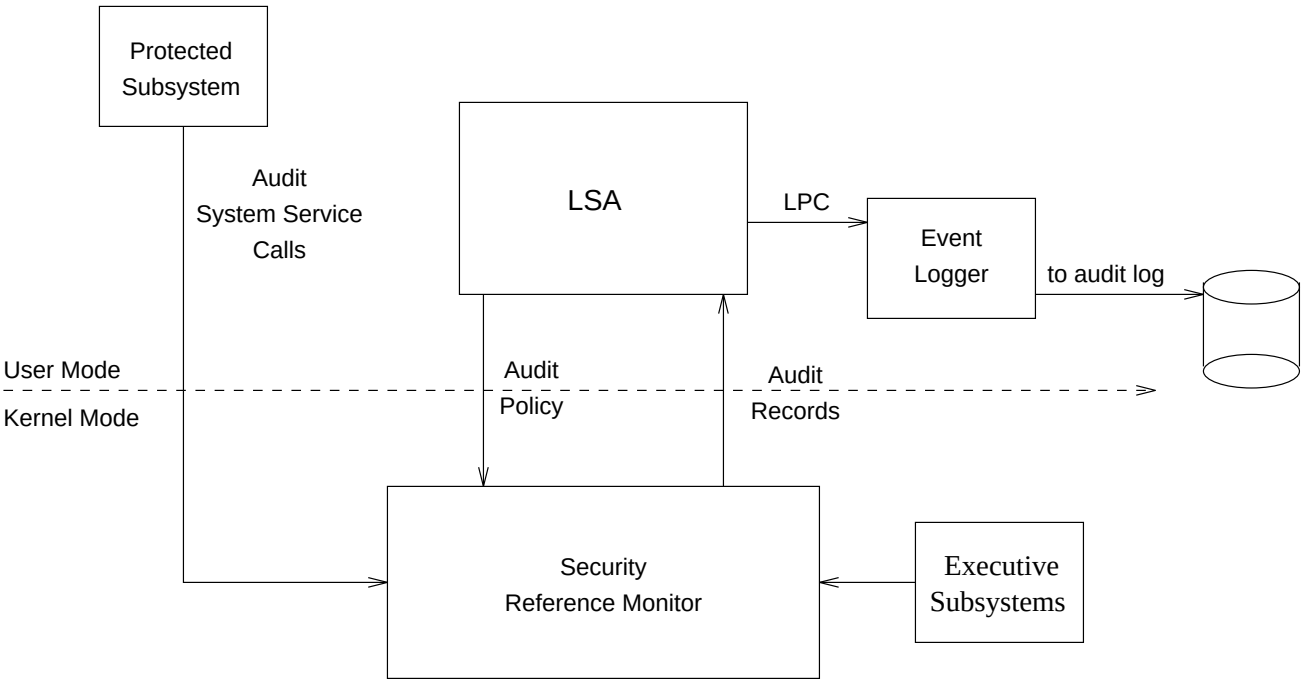


Figure 4.6. Audit Flow

4.1.4.4 Audit Generation

The SRM generates audit records for the set of event categories identified by the LSA. The SRM's auditing functions can be used by the executive as well as user-mode processes. However, user-mode processes calling audit system services to generate an audit record must have the Audit privilege. The privilege test is performed against the primary token of the process, not the impersonation token of the thread (if one exists). The overall flow of audit traffic is depicted in Figure 4.6.

4.1.4.4.1 Audit Policy The audit policy is defined by the set of parameters that the LSA sends to the SRM which controls auditing in the system. When an administrator changes the audit policy, the LSA updates its database and notifies SRM. The SRM receives the audit control information from the LSA in the following two pieces:

1. A control flag indicating if auditing is enabled
2. An array with each element representing an audit event category indicating that events in these categories will be audited. Each element will indicate if auditing shall be performed upon success, failure, or both. The event categories are:
 - System - indicates events which affect the security of the entire system or audit log (e.g., restart, shutdown).
 - Logon and Logoff - indicates the logon and logoff events

- Object Access - indicates events describing accesses to protected objects.
- Privilege Use - indicates events describing attempts to use security relevant privileges.
- Detailed Tracking - indicates events detailing subject tracking information (program activation, handle duplication, indirect object accesses, process exit).
- Policy Changes - indicates events describing high-level changes in security policy, such as assignment of privileges.
- Account Management - indicates events describing high-level changes to the security account database, such as user created.

The SRM puts audit records on an SRM internal queue to be sent to LSA. The SRM maintains a high water mark and a low water mark on its audit record queue. When the number of audit records on the queue reach the high water mark, the default of the system is to shutdown. This is to protect the system from discarding audit records which it will do when the audit queue(s) reach the high water mark. The system can be configured to shut down under this condition by default. If not configured this way and the queue reaches its high water mark, then the newest audit records will be discarded until the number of audit records in the queue equals the low water mark. In this situation, the SRM keeps track of how many records are discarded and generates an audit record which identifies the number discarded. This is the only type of audit record SRM generates which is not controlled by the LSA.

The audit records can move from the SRM to the LSA in two ways. If the audit record is small it can be sent as an LPC (see Section "Local Procedure Call Facility," on page 82) message. If the audit record is larger than the maximum LPC message size, the SRM allocates shared memory in the LSA and copies the audit record into the memory, passing a pointer to the record in the LPC message.

4.1.4.4.2 Object Access Audit Just as with access validation and privilege testing, it is the responsibility of other executive subsystems and servers to specifically request that the SRM generate an audit record for access to their objects. There are two ways to call the SRM to determine if an audit is necessary and if so, generate it. One way is to ask for this determination and possible generation at the same time as the request for access validation, by requesting a specific SRM system service which performs both. The other way is to request the SRM audit service separately, following the access validation request.

To determine if an audit should be generated, the SRM uses the token of the thread that requested the access, the desired access mask, the final determination of whether access is granted or denied, and the SACL. The audit algorithm evaluates each ACE in the SACL as listed below.

1. The ACE type is checked to see if it is either SystemAudit. If not, the ACE is skipped. Otherwise, processing continues to the next step.
2. The SID in the ACE is compared to the set of SIDs representing the subject. If no match is found, the ACE is skipped. Otherwise, processing continues to the next step.
3. The desired access mask is compared to the ACE access mask. If none of the accesses specified in the ACE were requested, the ACE is skipped.
4. The SuccessfulAccess and FailedAccess flags of the ACE are compared to the final determination of whether access has been granted or denied. If access was granted but the SuccessfulAccess flag is not set, or if access was denied but the FailedAccess flag is not set, the ACE is skipped. Otherwise processing continues to the next step.

5. Depending on the ACE type an audit is generated and processing continues with the next ACE.

4.1.4.4.3 Privilege Use Audit The SRM provides auditing functions to the executive and subsystems. These auditing functions audit privileged object accesses and the use of privileged system services, which are described in Section "Privilege Testing," on page 77. As in auditing object access, it is the responsibility of executive subsystems or servers to specifically call these SRM functions to generate privilege use audit records.

4.1.4.5 Logon Session Tracking

The SRM tracks logon sessions by keeping a count of the tokens associated with a logon session. When a user logs on, the LSA sends the SRM a message via LPC including the authentication ID (logon ID) assigned to the session. The SRM stores this authentication ID in its internal database and when a token is created for this session, or whenever this token is duplicated, the SRM increases the token count for the associated logon session. The SRM decrements this count when tokens are deleted. When a logon session's associated token count becomes zero, the SRM removes the authentication ID from its database and sends a message to the LSA that it has deleted this logon session from its database.

There is one occasion where the LSA sends a message to the SRM via LPC specifying to the SRM to delete a logon session from its database. This occurs during logon when the LSA has already called the SRM to create the logon session in its database, and the token for the session has not been created yet. At this point, some event can cause the logon to be denied and the LSA then calls the SRM to delete this logon session entry from its database.

4.1.5 Process Manager

The Process Manager handles all operations associated with processes and threads. It exports process and thread objects to user-mode processes and to other Executive subsystems; and provides functions relating to their creation, manipulation, and termination. The Process Manager is actually quite simple. Processes and threads are abstractions of the Windows NT microkernel which the Process Manager binds together, along with a handle table and primary token. The Process Manager interacts extensively with the Object Manager and the Virtual Memory Manager in order to provide the majority of its functions.

4.1.5.1 Process Definition

An Windows NT process consists of an address space, a handle table, a token (referred to as the primary token), and a set of zero or more threads—each represented by a thread execution block (TEB).

There is one handle table per process, which identifies the objects currently accessible to the process and its threads. The token contains the security relevant information about the process and its threads. The token is used by the security reference monitor to enforce the system security policy on the actions of the process or thread. Each thread execution block contains the information necessary for the scheduling and execution of the thread.

A process is implemented within the Windows NT Executive via a process object. The process manager maintains the information contained in the body of the process object, initializing it when the process is created and updating process-related information when requested. Pointers to the Process ID and Primary Token are the only security-relevant information maintained in the body of the process object.

The Process Manager handles requests from user-mode threads for the creation and deletion of processes and threads. When the creation request is made, the requesting thread may optionally specify the security attributes of the new process. If not specified, the new process receives the same security attributes as the creating process. Additionally, the creating thread must specify whether the new process is to receive a copy of the creating processes' handle table. If so, each handle is duplicated for the newly created process. Upon receiving a request for process creation, the Process Manager makes a request to the Object Manager to create a process object. After the creation by the Object Manager, the Process Manager initializes the process object body, and passes the process handle received from the Object Manager back to the requesting subject. At the completion of the process creation, there is no relationship maintained between the creating and created processes (i.e., there is no parent/child relationship). This allows processes to continue to exist even after the process that created them terminates.

In addition, the Process Manager provides functions to user-mode threads to request process information or to manipulate various process attributes. Many of these functions, similar to creation and deletion functions, are simply forwarded to other Windows NT subsystems. The remaining functions involve only the information in the process body. These functions include: requests for process identification information and requests for process memory operations.

4.1.5.2 Threads

The thread is the schedulable entity in Windows NT. A thread cannot exist outside the context of a process. A thread must be associated with one, and only one, process. A Thread Execution Block (TEB) contains all the necessary context information for a thread to run. Except in the case where the thread is impersonating another process, all security relevant information (i.e., ID and security attributes) of a thread are the same as those for the process to which it belongs and are contained in the process primary token. If the thread is impersonating another thread, then it possesses an impersonation token containing the relevant security attributes[17] of the thread being impersonated. Additional detail on impersonation can be found in Section "Impersonation," on page 70 and Section "Tokens," on page 70.

The Process Manager exported functions related to threads are similar to those provided for processes (e.g., creation, termination, query information) with the addition of functions for impersonation of and alerting of threads.

4.1.6 Local Procedure Call Facility

The Local Procedure Call (LPC) facility is an executive subsystem which implements interprocess communication between two threads in different processes. LPC may be used between threads in different modes (kernel-mode and user-mode) as well as in the same mode. The port object is the main data structure supporting the LPC mechanism. LPC exports the port object, providing functions to user-mode threads for manipulation. LPC supports three ways to pass messages: copying a message to a port, passing a message in

[17] The attributes are determined based on the level of impersonation

shared memory, and directly reading from and writing to a client's address space. Each method is described in this section. In the LPC context, the thread requesting the connection is referred to as a client and the thread receiving the connection is referred to as the server.

4.1.6.1 Port Object

During system initialization, LPC is called to create the port object type, which contains generic information about a port. Ports are not inheritable, cannot be permanent objects, cannot be exclusive, and do not implement the openif semantics upon the open request. (See Section "Executive Object Type Management Services," on page 58 for more details on these features.) There are two variations of the port object: a connection port and a communication port. A connection port is created by a server to allow clients to send connect requests. After a connection is established, two communication ports are created, one for the client and one for the server. LPC uses the port object to provide functions to implement the sharing of data.

4.1.6.1.1 Connection Ports A connection port is created by a server process by providing a name and a security descriptor to the LPC create function. Connection ports have names, making them visible to all Windows NT processes. These names are used as parameters in the calls to LPC made by processes to set up a communication channel with a server. The connection port name is placed in a directory in the global object name space (see Section "Global Name Space and Related Objects," on page 60) accessible to its clients. As long as a client has access to the directory, it can reference the port and request a connection. If an ACL is specified in the security descriptor to the create function, the connection port object will have an ACL. The only access associated with a port is PortConnect access. The body of the port object is zeroed by LPC and a flag is set in the port object to indicate a connection port. This flag is checked by LPC on connect calls, ensuring that the connect request can only be made to a connection port. Connection ports are associated with a queue (see Section "Synchronization Objects," on page 50) to hold messages sent to the connection port from a client. The server then receives a handle to the port.

4.1.6.1.2 Communication Ports Communication ports do not have names and are created by LPC as a result of the server accepting a connection request. For each connection request accepted, LPC creates a client communication port and a server communication port. These ports are zeroed by LPC and initialized to be a communication port. The client communication port is initialized with a flag set indicating it is a client communication port and pointers to the connection port and the server communication port. The server communication port is initialized with a flag set indicating it is a server communication port and pointers to the connection port and the client communication port. Communication ports are associated with a queue (see Section "Synchronization Objects," on page 50) to which the client or server queues its outgoing messages.

4.1.6.2 Client-Server Connection

After the connection port is created, the server remains in a wait state until a client issues a connect request to the connection port.

The client makes a connect request to the server by calling LPC functions referencing the connection port by name. The client will never get a handle to the server's connection port. In this connect request, the client specifies the Security QOS (see Section "Impersonation," on page 70) and the handle to a section object if a

shared section is being used to communicate. The connect request invokes an access check. However, it is the server process' decision to accept or reject a connection request, after an access check returns success. The server can choose to require a specific privilege, as LSA does on its port to register a logon process. The LPC create function then creates a communication port object for the client. Pointers to both the connection port and the security QOS parameters are stored in the client communication port. However, a handle to the client port is not yet inserted into the client's object handle table until the connect request has been accepted and is linked (the pointer field in the server communication port points to the client communication port and vice versa) and to the server's communication port. The LPC connect function then resumes the server by copying the connect request message onto the server's connection port's queue. The client is then put into a wait state. The connect function generates a unique message id and stores it and the id of the client in the message to the server. The connect function also stores the message id and pointer to the message to the server in the thread object of the client.

The server must call LPC to accept or reject the connect request. In accepting or rejecting a connect request, the server must include the ID of the message being responded to and the ID of the client which is to receive the reply. The LPC accept function verifies that the client specified in the accept function is waiting for a reply. Client-Server communication is based on the server responding to the client's request. A server cannot send a message to a client thread which is not waiting for a message. This is accomplished by comparing the message ID passed in the accept call by the server to the ID in the thread object of the target client whose ID was specified by the server. The accept function will succeed only if a match is found. When the server succeeds in accepting the request, LPC creates a communication port for the server. The client and communication ports are linked together, pointing to each other and to the connection port. A handle to the server communication port object is returned to the server and is inserted in the server's object handle table.

When the server is ready to communicate with the client thread, it calls LPC to complete the connection request. LPC then resumes the client and passes the handle to the client communication port to the client and inserts it into the client's object handle table. The client port handle is valid only to the client process and the server port handle is valid only to the server process. The client may now send messages to the server. Figure 4.7 shows a client-server connection.

4.1.6.3 Copying a Message to a Port

Copying a message to a port is the most common form of message passing used by LPC. Each communication port object contains a queue of fixed-size message blocks. LPC copies a client request from the client's address space into one of the message blocks in the server's communication port object. After a context switch from the client to the server process, a server thread copies the message into the server's address space and processes it. To reply, the server sends a message back to the client's communication port.

4.1.6.4 Passing a Message in Shared Memory

Shared memory is an alternative and faster way to exchange data and messages between clients and servers. It is also used when a client needs to pass messages that are greater than 256 bytes since this message is too large to fit in the server communication port's message queue. The section object (see Section "Section Objects and Shared Memory," on page 64) is used to implement the shared memory technique of passing messages. A section object is a block of shared memory that LPC makes visible in the address space of the client and server. LPC supports mapping the section, however, creation and management of the section

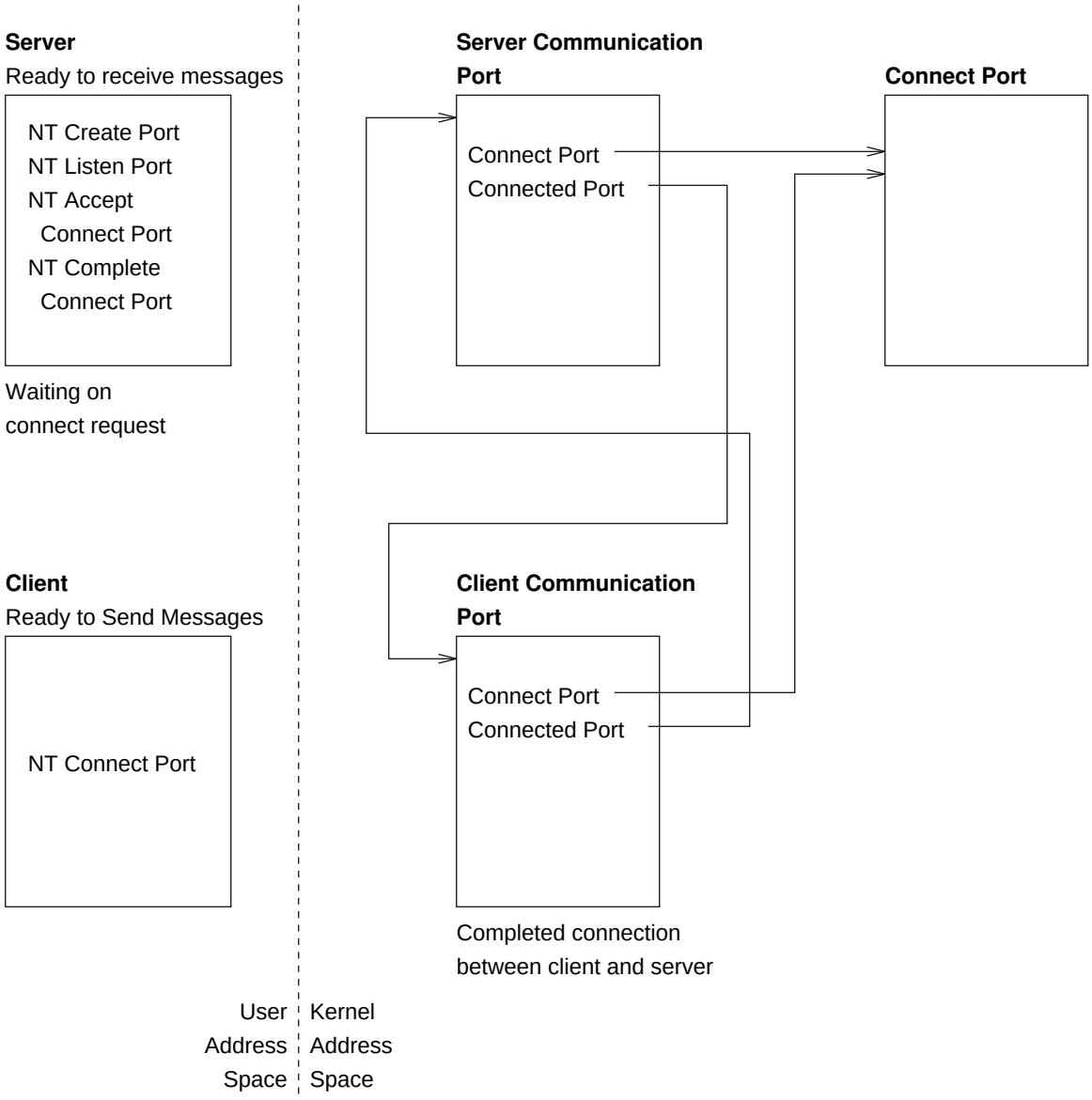


Figure 4.7. Client-Server Connection

object is left up to the client and server. It is the responsibility of the client and server to validate pointers into the section. LPC does not do this for them.

The Virtual Memory Manager provides functions to create, allocate, and map the section object into the caller's address space returning a handle to the section object. The process initiating a request (typically the client) is responsible for creating and managing the shared section. The client places the message in the section object and sends a pointer to the section object and the size of the section object to the server's communication port. After the client creates the section object, it passes the handle to LPC when requesting a connection to a port. LPC maps the section object into the client's and server's address spaces. In order for LPC to perform the mapping, it must be passed a data structure describing the section to the client.

LPC calls Virtual Memory Manager to map the section object into the client's address space. As a result, the base address of the section in the client's address space, the view size, and section offset are stored in the client's port view structure. When the server is ready to accept the connect request, it passes a data structure to the LPC accept connect function. LPC returns the base address at which the section is mapped into the server's address space with the assistance of a Virtual Memory Manager. The server will use this base address to validate address pointers into the section passed by the client. This base address is stored in the server's communication port and in the connection message for returning to the client.

When the server calls LPC to complete a connect request, the client is resumed by the LPC complete function. The client now has a section mapped into its address space and a data structure containing the base address in client address space, the base address in server address space, the size, and the offset. The server has the base address of the section in its own address space and the size of the view. This information is used by the server to protect itself from faulty pointers passed by the client. Figure 4.8 shows client-server communication using a section object.

4.1.6.5 Direct Communication

LPC provides two functions for reading from or writing to a client's address space. This is used when a server wants to read or write data larger than can fit in a section object. A small message must first be sent to synchronize the message passing.

The client passes parameters in the LPC calls that describe the buffers that may be read from or written to by the server. These parameters, which include the buffer address and length, are kept by LPC which provides services at the server's request to read from or write to these buffers. The server calls LPC providing an index into the list of buffers indicating which of the buffers to read from or write to. LPC will then read from or write to the appropriate buffer for the server.

4.1.6.6 LPC Impersonation

The LPC mechanism is used to implement impersonation (see Section "Impersonation," on page 70). The SQOS parameters identify if the client will allow the server to impersonate or not. Impersonation can only occur if the SQOS parameters allow it. The client sends the server its SQOS parameters (see Section "Impersonation," on page 70) when the client makes a connection request to the server. If the SecurityContextTracking SQOS parameter is static, LPC duplicates the token and saves it when connection is requested. If the SecurityContextTracking parameter is dynamic, LPC keeps a pointer to the client thread's token when connection is requested. This token information is used by LPC if the server makes an LPC impersonate call

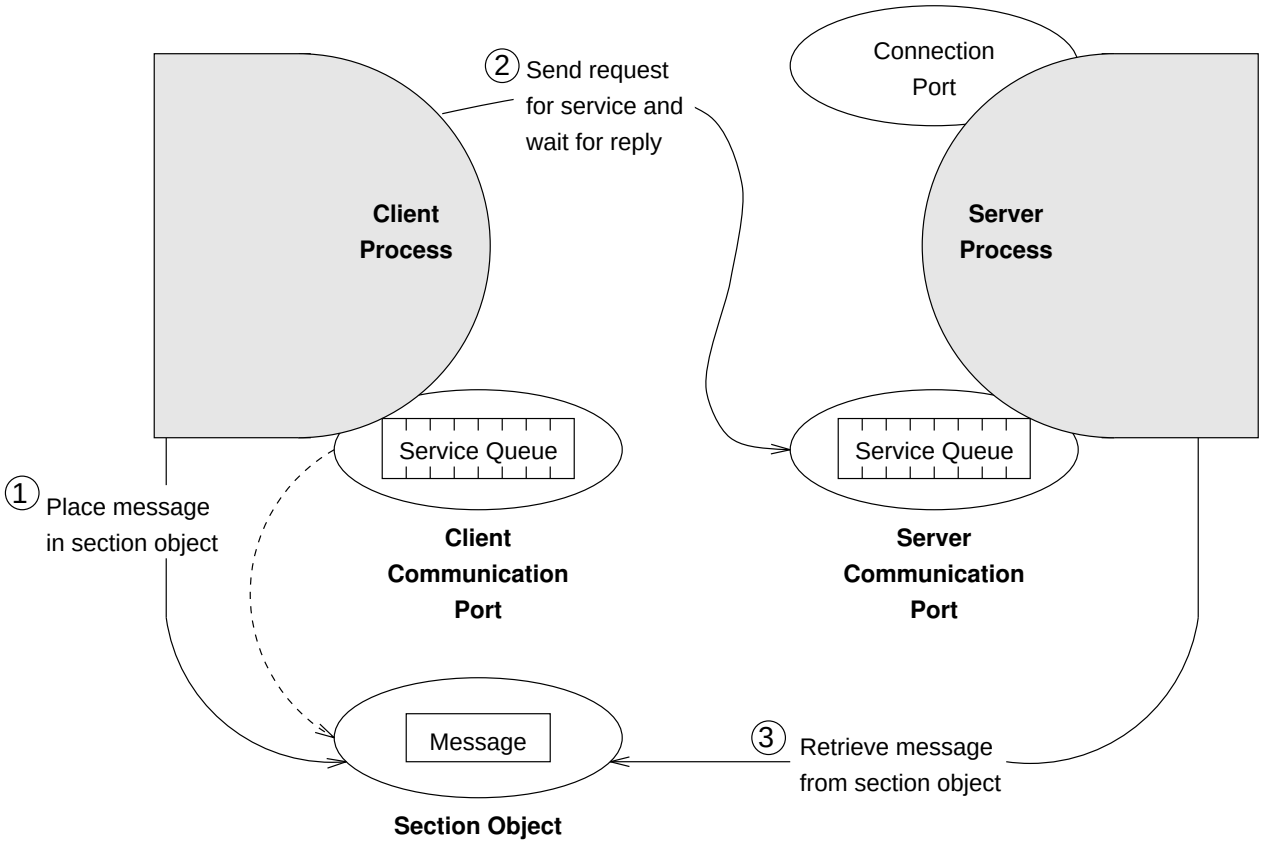


Figure 4.8. Communication using Sections

to impersonate the client associated with a specified server communication port. At that time, LPC assigns the duplicated token it saved (static tracking) or the pointer to the client thread's token it saved (dynamic tracking) to the server thread making the impersonate LPC call. This server thread is now impersonating the client and has an impersonation token.

If this impersonation token is dynamic, then all modifications made by the client to its token are effective to the server. During this connection, the impersonating thread can obtain a handle to its impersonation token by calling an executive service which allows a thread to open its token. This service returns a handle to the token. When an impersonating thread makes this call, LPC will duplicate the impersonating thread's token and return a handle to the duplicated token if dynamic tracking was specified. Otherwise, if static tracking was specified, a handle to the impersonating thread's token is simply returned (token has already been duplicated).

This process does not allow server modifications to an impersonation token to impact the client, even when dynamic tracking was specified, since LPC always returns a handle to a duplicate copy of the client's token. In summary, dynamic tracking allows for client modifications to an impersonation token to be effective to the server (not vice-versa) only as long as the token is not opened (modifiable) by the server.

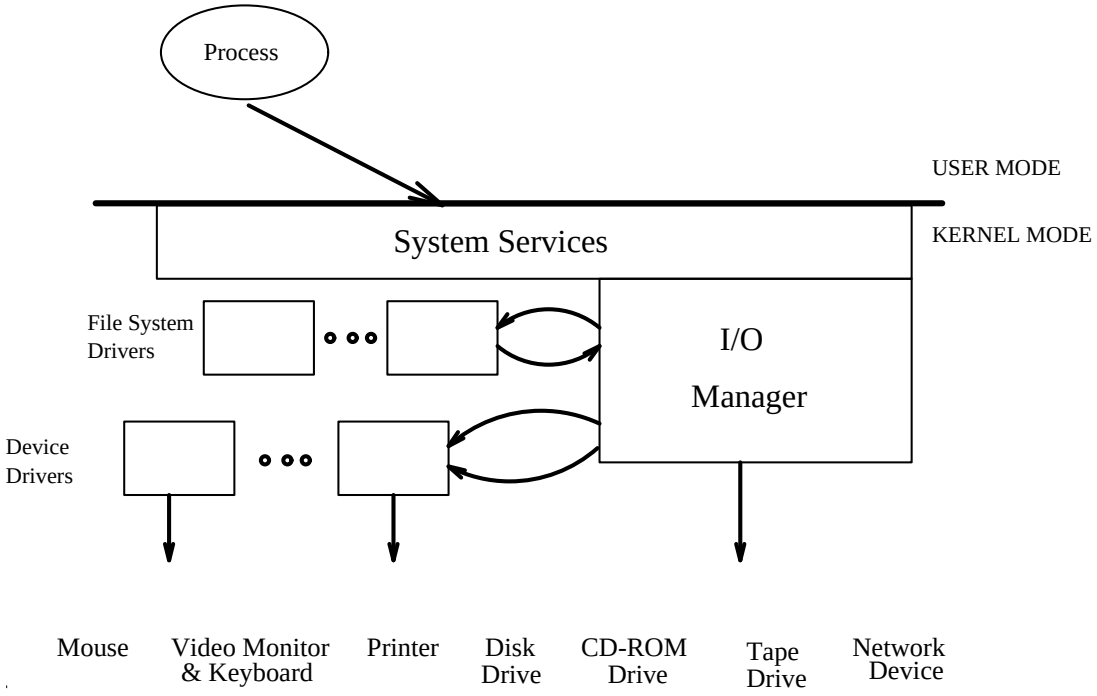


Figure 4.9. The I/O Subsystem

Impersonation is terminated when the server impersonating thread calls the executive to change its token to null and all handles gained during impersonation are kept.

4.1.7 I/O Subsystem

The Windows NT executive's I/O subsystem processes I/O requests from user-mode and kernel-mode threads. As shown in Figure 4.9, the I/O subsystem supports a layered driver model consisting of *Device Drivers (DD)*, *File System Drivers (FSD)*, and the I/O Manager. In some cases, the I/O Manager processes an I/O request by locating the appropriate device and calling its device driver, which directly controls the physical device. This is the case for *single-layered* device requests such as the keyboard. Most I/O operations are not this direct, requiring more than one driver to be called to process the I/O request. A file system driver is a *layered* device driver because an FSD depends on support from the lower-level device drivers. For example, an I/O request to an FSD, such as an open file request, involves support from both the FSD, considered to be a higher-level driver, and the lower-level DD on which the file resides, such as a disk drive. Windows NT supports device drivers for five different file systems: the FAT file system, the NTFS, the CDFS, the NPFS, and the MSFS.

The I/O subsystem manages numerous objects created for the I/O subsystem by the Object Manager, However, many of these objects are not exported outside of the executive. Any exported I/O subsystem objects are exported as file objects. *File objects* are created by the Object Manager when a thread opens an I/O subsystem object. File objects have hierarchical names, are protected by object-based security, support synchronization, and are manipulated by object services. The Object Manager creates the file object and

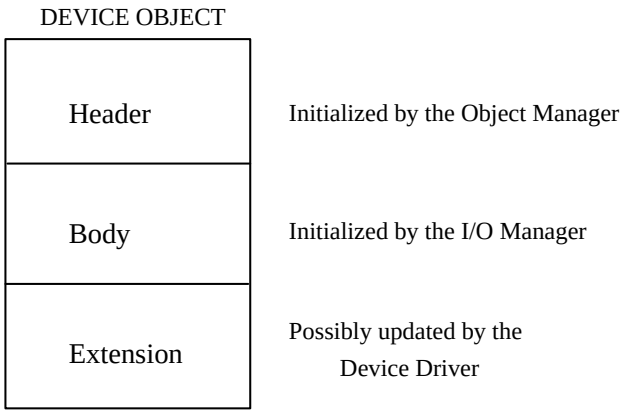


Figure 4.10. Device Object

initializes the header with the hierarchical name. The I/O subsystem initializes the file object body and returns a *file handle*, which is subsequently used to refer to the file object. From user mode, there is no other way to access a device, or a file on a device, other than through file object handles supplied by the I/O subsystem (see Section "Layered Device Drivers," on page 92).

A *device object* is not exported outside of the executive. A device object represents a physical, logical, or virtual device on the system and describes its characteristics, such as the alignment it requires for buffers and the location of its device queue. Some device objects are created when the file system which manages the device is loaded; most device objects represent volumes which are created when referenced and are never loaded (such as a disk volume). The Object Manager creates the device object and initializes its header. The I/O Manager initializes the device object body and the device driver may initialize an extension portion of the object (see Figure 4.10). A *device object extension* is a data structure whose format is driver-defined. It is used to maintain device state information, to provide storage for any microkernel-defined objects used by the driver, and to hold any data the driver must have resident in memory in order to carry out its I/O operations. Most drivers find it convenient to store pointers to their device objects in the device extension; higher-level drivers almost always store pointers to the next lower-level driver's device objects.

A *driver object*, also not exported outside of the executive, represents an individual driver in the system. The Object Manager creates the driver object and initializes the object header for the I/O Manager when a driver is loaded into the system. Once the object has been created and initialized by the Object Manager, the I/O Manager calls the device driver to look for the actual device. If no device exists, the driver object is deleted. If a device does exist, the driver initializes the driver object body which has three main parts: pointers to devices operated by this driver, a set of entry points (addresses) for driver routines, and a function table. As shown in Figure 4.11, the object body of a driver object maintains pointers to the devices it operates. For mass storage devices (disk, tape, Compact Disk Read Only Memory (CD-ROM)), a device object may not be present if the device is not mounted. For example, if a particular disk volume is not mounted, a device object will be created for the volume as part of the mount operation, and linked to the list pointed to by the file system driver object. The device driver also initializes the driver object with a set of addresses for certain standard routines that are implemented by every Windows NT device driver, and potentially a set of optional routines that may also be implemented. Single-layered device drivers are discussed more fully in the next section.

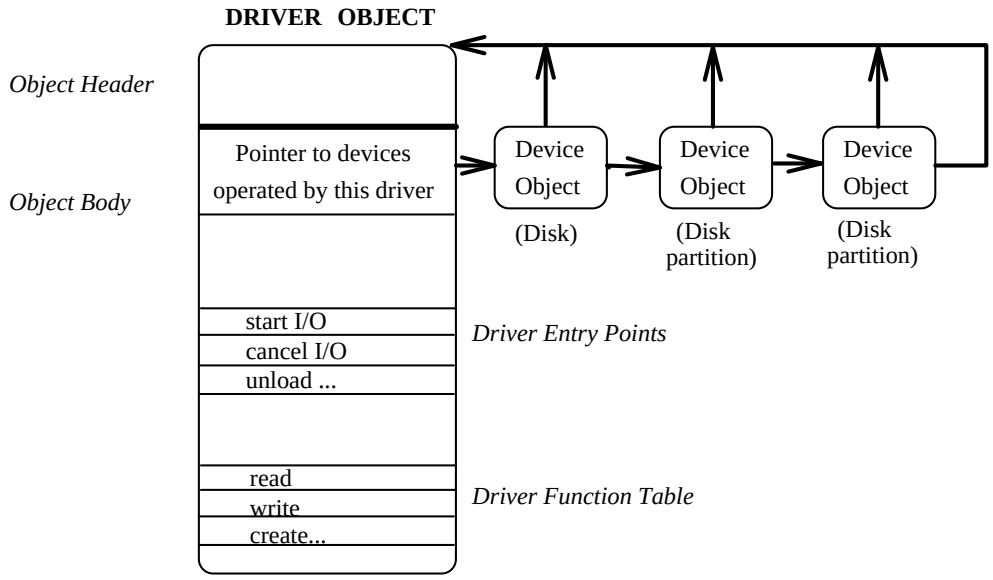


Figure 4.11. Driver Object

Device drivers (including file systems) are designed to be dynamically loaded into the Executive, and may be loaded during system initialization, during subsequent processing, or not at all, depending upon when (and whether) they are needed.

4.1.7.1 Single-Layered Device Drivers

Each Windows NT driver implements certain standard routines. For example, every driver has at least one Dispatch entry point but does not necessarily implement the StartIo routine. Table 4.6 identifies the most significant standard driver routines, although a driver might additionally implement optional routines, i.e., other than the standard driver routines.

The I/O Manager provides asynchronous I/O support so that the originator of an I/O request can continue executing, rather than wait for its I/O request to complete. A large data transfer request may be split into smaller requests or I/O requests may be reordered. As a consequence, Windows NT drivers do not necessarily process I/O requests in the same order they were sent to the I/O Manager. Instead, drivers respond to a request as it is passed to the driver's routines by performing whatever routine-specific operations are necessary to satisfy the current request.

The I/O subsystem is *packet driven*, which means that every I/O request is represented by an *I/O Request Packet (IRP)*. An IRP is a data structure that controls how the I/O operation is processed at each stage along the way. The I/O Manager accepts I/O requests, creates IRPs to represent the requests, routes the IRPs to the appropriate drivers, tracks the IRPs until they are completed, and returns the status to the original caller. A driver receives an IRP, performs the operation the IRP specifies, and passes the IRP back to the I/O Manager. The I/O Manager may subsequently forward the IRP to another driver for additional processing.

Entry Point	Function
Initialize, Reinitialize	Creates objects that the I/O Manager uses to recognize and access the driver. A driver may have a reinitialize routine if it needs to initialize itself in stages.
Dispatch	The main functions that a device driver provides. All drivers must have at least one dispatch routine in order to handle I/O requests. Most dispatch routines handle the following types of I/O requests: create, cleanup, close, read, write, control, status, flush buffers, shutdown.
StartIo	Used to initiate a data transfer to/from a device. This routine is not required since a driver can set up and manage its own queue of IRPs.
ISR	Drivers for physical devices that generate interrupts must have an ISR; FSDs do not have an ISR.
Deferred Procedure Call (DPC) routine	Drivers that have an ISR should also have a DPC in order to complete interrupt-driven I/O operations. A DPC initiates I/O completion and starts the next queued I/O operation on a device. See Section "Microkernel," on page 49 for a discussion of DPCs.
Cancel	Drivers in which IRPs might remain queued for an indefinite interval must have one or more Cancel routines to complete user-cancelled I/O requests.
IoCompletion	Higher-level drivers that allocate IRPs to send requests to lower-level drivers must have an IoCompletion routine.
Unload	If a driver can be loaded or replaced dynamically, an Unload entry point must be defined in order to free any resources that the driver may have allocated. A driver that is part of the base system and loaded at initialization, in general, does not have an Unload routine and is not unloaded.

Table 4.6. Device Driver Entry Points

Each IRP has a *fixed part* and one or more driver-specific *I/O stack locations*. The fixed part, or header, contains information about the original request, such as the caller's parameters, the address of the device object on which a file is open, etc. The fixed part also contains an *I/O status block* in which drivers set information about the status of the requested operation. The I/O stack location is used by the I/O Manager to set driver-specific parameters, such as the particular operation requested and the context of the operation. As an IRP is processed through each driver's set of standard routines, each routine can access that driver's I/O stack location in the IRP, thereby reusing the IRP at each stage of the driver's operation. In addition, higher-level drivers can create or reuse IRPs to send requests down to lower-level drivers, if needed. Note, however, that higher-level drivers do not directly call lower-level drivers; instead, the higher-level driver calls the I/O Manager, which in turn calls the lower-level driver. A higher-level driver in a chain of layered drivers can only access its own and the next lower-level driver's I/O stack locations in any IRP. This is because a driver would not know how to interpret the information on a higher-level stack. Such a driver must set up the I/O stack location for the next lower-level driver. The lowest-level driver in a chain of layered drivers can access only its own I/O stack location in any IRP. While active, each IRP is stored in an *IRP queue* associated with the thread that requested the I/O. This allows the I/O Manager to find and delete any outstanding IRPs if a thread terminates or is terminated with outstanding I/O requests.

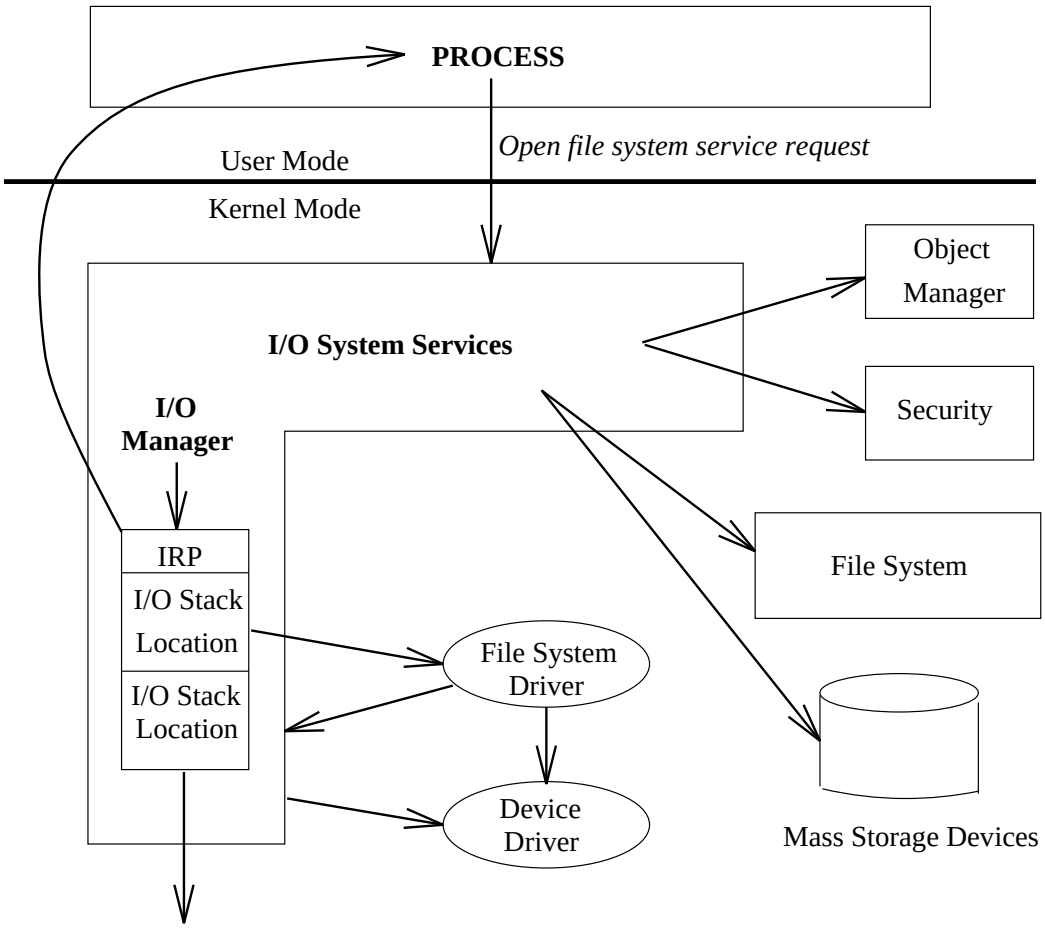


Figure 4.12. Opening an Windows NT File

4.1.7.2 Layered Device Drivers

As previously described, a layered device driver requires more than one driver to be called to process an I/O request. File system drivers (FSDs) are layered drivers because they depend on support from the lower-level drivers for the device on which the target file resides.

In Windows NT, processes and threads perform I/O on file objects, manipulating them using *file handles* which refer to executive file objects. To open a file, a thread supplies the file name and the type of access requested to an Windows NT system service, which ultimately returns the file handle to the requesting thread. The I/O subsystem and the Object Manager work together to handle the open file request as described in the following paragraphs (see Figure 4.12).

Step 1. An open file request, *NTOpenFile*, launches object name lookup in the Object Manager. For example, suppose a process is attempting to open the file `\device\harddisk2\partition3\foo\tmp.dat`.

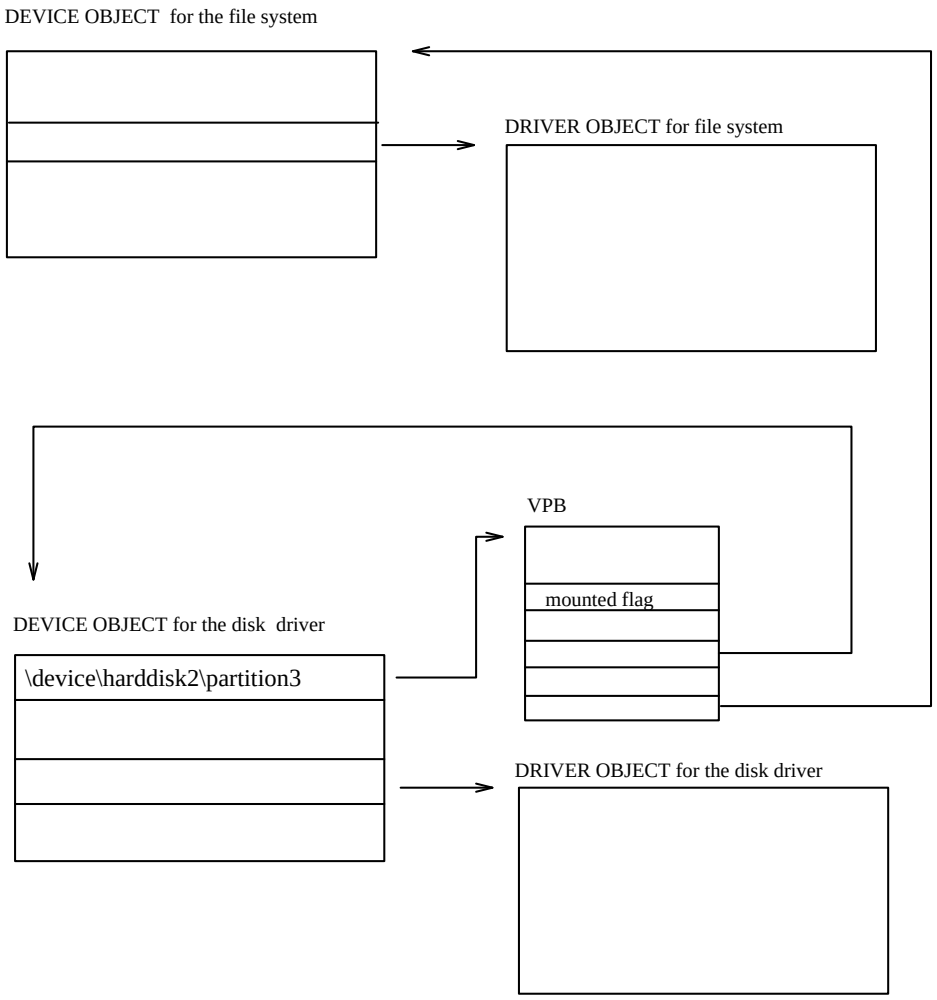


Figure 4.13. Device and Driver Objects

The Object Manager processes the hierarchical name and determines that the I/O subsystem handles objects with the name `\device\harddisk2\partition3`. The Object Manager passes the I/O subsystem parse routine a pointer to the device object with that name.

Step 2. The I/O subsystem calls the SRM for an access check on the device object for `partition3`. If the access check is successful, the I/O subsystem checks to see if the device is mounted, which is indicated by a flag in the Volume Parameter Block (VPB). If the volume is not yet mounted, the I/O subsystem suspends the open request, and calls each of the registered, active file system drivers. Each driver, in turn, examines the device object for `partition3` to see if it recognizes the format of the partition (see Figure 4.14. Once a file system driver recognizes the device's format, it creates a device object for the file system and connects it to the VPB in the partition device object and sets the mounted flag. Recall that device objects point to their driver objects and vice versa (see Figure 4.11). Once the device has been mounted, the I/O subsystem resumes the open request.

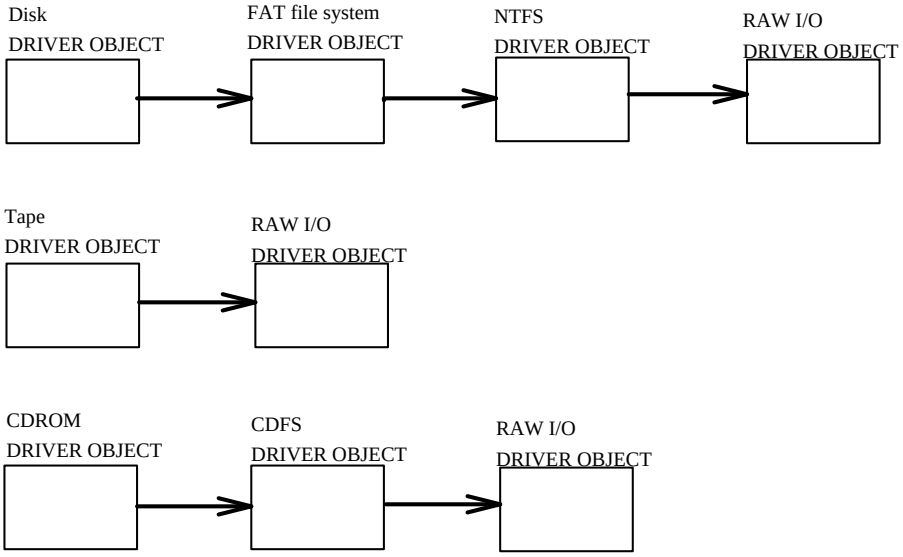


Figure 4.14. Mass Storage Devices

The I/O subsystem calls the Object Manager to create a file object. The Object Manager creates a file object and returns it to the I/O subsystem. The I/O subsystem inserts a descriptor of the rest of the pathname (``\foo\tmp.dat'') in the file object body.

Step 3. The I/O Manager allocates memory for and initializes an IRP for the open request, which is a create request to the driver, and includes a pointer to the file object in the IRP. The I/O Manager calls the file system driver, passing it the IRP and a pointer to the partition device object.

The file system driver accesses its I/O stack location in the IRP to determine which operation to perform, checks parameters, determines if the requested file is in the cache, and, if not, sets up the disk driver's I/O stack location in the IRP. Both drivers, the file system driver (the higher-level driver) and the disk driver (the next lower-level driver), process the IRP and complete the requested I/O operation, calling kernel-mode support routines supplied by the I/O Manager and other Windows NT executive components. In particular the SRM is called by the file system driver to check access first on "\foo" and then on "\tmp.dat".

When processing is completed, the drivers return the IRP to the I/O Manager with the I/O status in the IRP indicating whether the operation succeeded or why it failed. The I/O Manager gets the I/O status from the IRP, so it can return Windows NT status information to the original caller, and subsequently frees the completed IRP. If the operation was successful, the Object Manager inserts a handle to the file object in the calling process' handle table and returns status information to the caller.

4.1.7.3 File Systems

In the evaluated configuration, Windows NT supports the following file systems:

- Windows NT File System (NTFS)

- FAT file system
- CDROM file system (CDFS)
- named pipe file system (NPFS)
- mailslot file system (MSFS)

NTFS, the FAT file system, and CDFS are backed by disk files which reside on mass storage devices. Each of these file systems has a device object that is created when the file system is loaded. In order to access NTFS, FAT, or CDFS files and directories, a thread must have traverse access to the device object that represents the respective file system. Thus, the access control list on each of these device objects is established when it is created and is set to traverse access for world, and read/write access for administrators.[18] The file system device object is exported outside the executive as a file object, with the access control list. Subsequently, file objects may be created for individual file system objects (files and directories). File objects for FAT and CDFS files and directories do not have a pointer to a security descriptor in the object header. Instead, the protection for FAT and CDFS files and directories is based on the access control list on the file object that represents the file system device object. File objects for NTFS files and directories, on the other hand, do have a security descriptor (see Section "Access Control List Data Structure," on page 72).

NPFS and MSFS are represented within the executive by a device object. However, these file systems are not backed by disk files, thus, when a named pipe or mailslot is opened, a file object is created which represents an *instance* of the named pipe or mailslot. A parse method is provided for named pipe and mailslot objects (see Section "Object Manager Exported Objects," on page 60) for a discussion of parse methods. Each of these file systems is discussed in detail in this section.

4.1.7.3.1 Cache Manager The Cache Manager is a component of the executive that provides system-wide caching services. In Windows NT, the file systems that are backed by disk (NTFS, FAT, CDFS) use the services of the Cache Manager and the Virtual Memory Manager (see Section "Virtual Memory Management," on page 63) to implement memory-mapped I/O.

The Virtual Memory Manager (VMM) implements *section objects* which it uses to represent *file streams*. A file stream is a linear stream of bytes associated with a file object. Examples of file streams include the data of a file, the ACL of a file, a directory, or any other file metadata, depending on the particular file system. File streams are created, deleted, and maintained by each file system. A file system must identify each supported stream as cached or non-cached. When a file system calls the Cache Manager to initiate caching of a file stream, the Cache Manager calls the VMM to map all or a portion of the stream into section objects. A section object data structure is used to identify which section objects in the cache are used by the file object. A pointer to this data structure is stored in the file object body.

For larger streams, the Cache Manager may subsequently find it necessary to map additional portions of the stream on an as-needed basis. To keep track of which portions of a file stream are currently cached, the Cache Manager uses local (private) data structures called *cache maps*. For each stream being cached, the Cache Manager maintains a single *shared* cache map which describes an initial portion of the file stream mapped for common access via all file objects for this stream. For each file object through which the cached stream is being accessed, the Cache Manager also maintains a *private* cache to buffer read-ahead blocks. A pointer to the private cache is also stored in the file object.

[18] Although a file system is not supported for tapes, a device object does exist for the tapes. The device object has an access control list that is set to traverse access for world, and read/write for administrators.

When a thread opens and uses a file, the file system tells the Cache Manager to create unnamed section object(s) and maps the file into the section(s). As the thread uses the file, the Virtual Memory Manager brings accessed pages into the section objects from disk and flushes them back to disk during paging. Subsequent accesses will look first in the cache. When an attempted access results in a cache miss, a page fault results causing the Memory Manager to recursively call back to the file system with a non-cached I/O request.

4.1.7.3.2 Windows NT File System (NTFS) The Windows NT File System (NTFS) is a new file system designed for Windows NT. NTFS is managed by a loadable driver that can be added or removed from the system as needed. As previously described, NTFS implements memory mapped I/O via the Cache Manager and the Virtual Memory Manager. Unlike CDFS and the FAT file system, NTFS also uses the *log file service* to log transactions. The log file service is a set of library routines within NTFS that are used for recovery of disk information in the event of a system failure. Basically, a log file is kept of all disk writes which can subsequently be used to recover an NTFS-formatted disk.

NTFS implements two kinds of file system objects: files and directories. Files and directories are implemented the same way in NTFS, i.e., they have the same structure except that a file stores the file data and a directory stores a list of file names contained in the directory. This is explained in detail in the next section. Both types of objects are exported outside of the file system as a file system object (including the optional security descriptor), as previously described.

4.1.7.3.2.1 NTFS Volume Layout The structure of NTFS is based on a *volume*. A volume corresponds to a logical partition on a disk that is created when the disk, or part of the disk, is formatted for NTFS. Whether a physical disk has only one volume or several volumes, NTFS handles each volume independently of the others. A volume consists of a series of *files*, plus any additional unallocated space remaining on the disk partition. An NTFS volume consists *only* of files and unallocated space. In other words, all file system data is stored as ordinary files (reserved for system use only) on the volume, which is not the case for the FAT file system, which reserves some areas of the volume for file system use.

NTFS uses *clusters* as its fundamental unit of disk allocation and can adjust the cluster size to make best use of the size of the disk being formatted. When a user formats an NTFS volume, the format utility establishes the volume's *cluster factor*. The cluster factor defines the number of physical sectors per cluster and is always a power of two. Internally, NTFS addresses everything by *cluster number* and is unaware of the sector size of the device. This maintains a level of independence from physical sector size, which can vary across disks. NTFS refers to physical locations on a disk by using *Logical Cluster Numbers (LCNs)*. LCNs are simply a numbering of all clusters on a volume from the beginning of the volume to the end. To convert an LCN to a physical disk address, NTFS multiplies the LCN by the cluster factor to yield the physical byte offset on the volume, which is required by the lowest-level disk driver.

NTFS maintains one central file per volume, called the *Master File Table (MFT)*, which is the heart of the NTFS volume structure (Figure 4.15). The MFT contains one *file record* for each file on the volume, including a record for the MFT itself. Each NTFS volume also contains a set of system files that are used to implement the file system structure, as shown in the figure. The remaining files on a volume are user files and directories. Files on an NTFS volume are identified by a *file number* which is derived from the position of the file record in the MFT, and a *sequence number* which is incremented each time a file record position is reused. The sequence number allows NTFS to perform internal consistency checks as it accesses files in the MFT. The file number is 6 bytes in length, and the sequence number is 2 bytes, resulting in a 64-bit number, called the *file reference*, which is used to uniquely represent a file. An MFT entry is allocated for

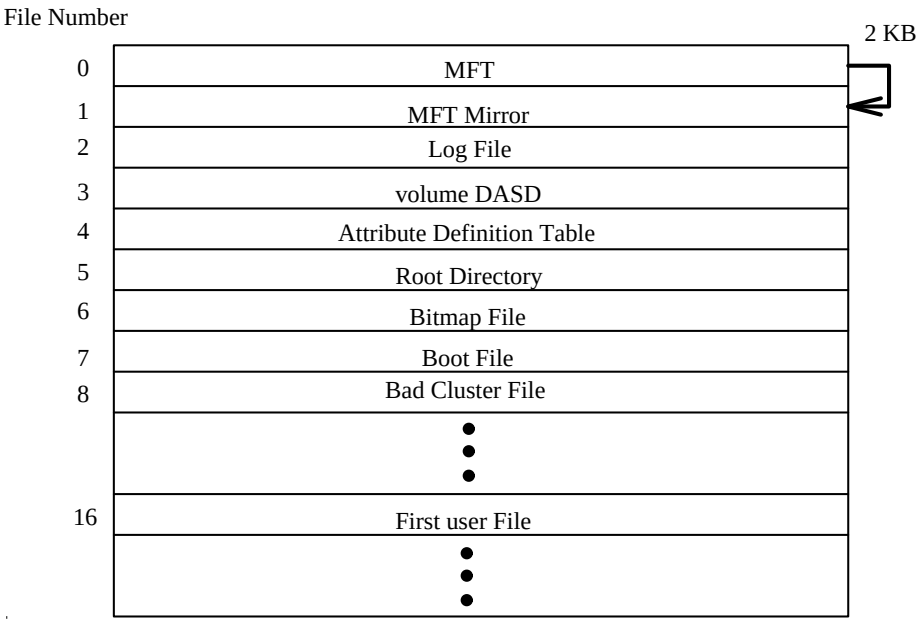


Figure 4.15. Master File Table

a new file by setting its corresponding bit in the MFT BITMAP attribute to the *allocated* state. An MFT entry is deallocated when a file is deleted by setting its corresponding bit in the BITMAP attribute to the *free* state.

Other relevant system files stored in the MFT include a replicated MFT file record, called the *MFT mirror*, in case the first MFT entry cannot be read. The *log file* is used to record all operations that affect the NTFS volume structure, including any disk writes and copy or rename commands that alter the directory structure. Two types of information are recorded in the log file: redo information and undo information. Redo information tells NTFS how to re-apply an operation if a system crash or disk failure occurs. Undo information tells NTFS how to reverse an operation that did not complete due to an error code. Currently, redo and undo information is only stored for changes to system files, not user files. The *root directory* entry records an index of all the files stored in the root of the NTFS directory structure for that volume. The *bitmap file* records the allocation state of the volume. Each bit in the map represents a single cluster on the volume, identifying whether the cluster has been allocated to a file. The *boot file* stores the bootstrap information and the *bad cluster file* records any bad spots on the disk volume.

An NTFS file is defined as a set of one or more attributes, for example, a file name attribute, or a security descriptor attribute, as listed in Table 4.7. An attribute consists of a type, length, and value and is stored as a stream of bytes in the MFT. In actuality, the NTFS model does not view a file as only a repository for textual or binary data but views a file as a collection of various attributes, one of which is the data it contains.

For small files, the attributes, including the file's data, fit into a single table entry. When stored directly in the MFT, the attributes are called *resident attributes*. For a small directory, the index root attribute stores the entire list of file names contained in the directory. The MFT entry (the file record) for a small file and a small directory is shown in Figure 4.16.

System Defined Attribute	Description
Standard Information	Includes the file type and time stamps (time of creation or last modification).
File Name	The file name in unicode characters. A file can have multiple file name attributes.
Security Descriptor	Security data structure that protects files from unauthorized access. It specifies who owns the file and who can access it.
Data	The file's contents; there can be more than one data attribute, for example, if multiple streams are defined. The data attribute also stores size information about the file and its attributes,
Index root, index allocation, bitmap	Three attributes are used to implement a storage scheme for large directories, whose entries cannot fit directly into the MFT. The directory entries are stored in separate disk allocations organized in a b+-tree format.
Attribute List	Lists all the attributes comprising a file and the file record in which they are located. Used when the number of attributes is large requiring two or more rows in the MFT to represent the file.

Table 4.7. System Defined File Attributes

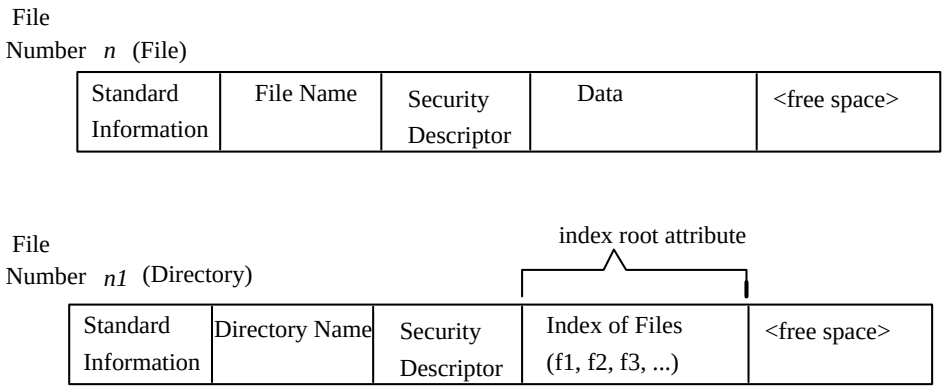


Figure 4.16. Small File and Directory MFT Entry

Not all files or directories can be stored into a 1 or 2 KB data structure. For large attributes, such as the data attributes of a large file, NTFS allocates separate areas on the disk called *file runs* or *file extents* to store the information. Attributes that exist in separate file runs are called *nonresident attributes*. The MFT entry for the file contains the information necessary to locate file runs. The fact that an attribute is resident or nonresident is completely transparent to the thread accessing the file. The MFT entry with two file runs is shown in Figure 4.17.

When a file run must be allocated, NTFS keeps track of the various pieces of the file by using *Virtual Cluster Numbers (VCNs)*. While LCNs represent the sequence of clusters on the entire disk from 0 to n, VCNs number the clusters belonging to a particular file from 0 to n. Figure 4.17 illustrates the VCN numbering for

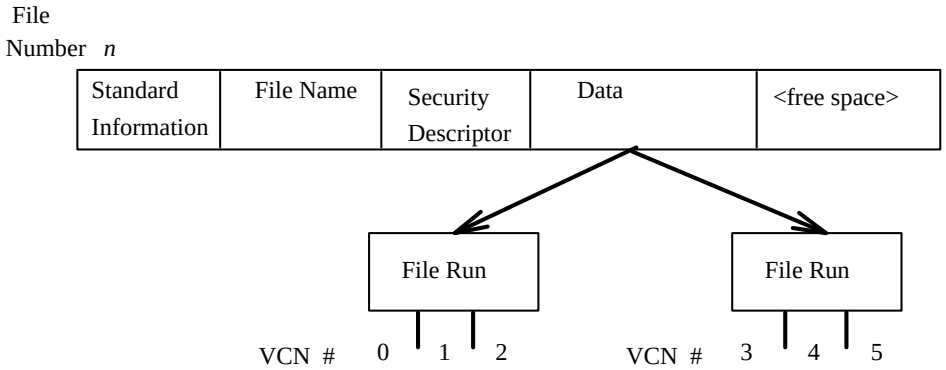


Figure 4.17. Large File MFT Entry

a file with two data runs, where each run has three clusters. If the file had more than two runs, the clusters would continue to be assigned a VCN starting with six. Any file attribute, not just the data attribute, can be nonresident. If a particular file has too many attributes for a single MFT entry, additional entries may be allocated.

A file directory is an index of file and subdirectory names, i.e., a collection of names organized in a particular way for quick and easy access. Associated with each name is a file number, which is an index into the MFT. NTFS uses the file number to locate the MFT entry for the file or subdirectory associated with the name. Since NTFS knows the root directory for each volume (file number 5), it is able to walk the directory structure finding file names and associated file numbers (and MFT entries) for all objects on a volume. It is possible for a given file or directory to have multiple names (though only one MFT entry). The APIs for NTFS allow a program to place multiple names (in possibly multiple directories) each with the same file number. This in effect provides multiple names, also called *hard links*, to the same object. In such cases, the object and its associated MFT entry would not be deleted until all names for that object are deleted.

An MFT entry for a large directory looks similar to the entry depicted in Figure 4.17 with one or more runs containing the names of files in the directory. The index root attribute contains the entire sorted list of files, or points to the runs that contain the entries, sorting the list in a b+ tree manner for quick searching. The index allocation attribute keeps track of the mappings between the VCNs of the disk allocations that make up the directory file and the LCNs where the files reside on the disk. The bitmap attribute keeps track of which VCNs in the index allocation are in use and which are free.

4.1.7.3.2.2 File Processing As previously described, files and directories are exported as file objects. Threads create or access a file or directory via a handle to the file object. By the time requests reach NTFS, the I/O Manager has transformed the file handle into a pointer to a file object. NTFS uses the file object to record the location of the file or directory on disk, so that the file can be located quickly on subsequent accesses.

NTFS is called with a pointer to a file object. As shown in Figure 4.18, a file object points to a *Stream Control Block (SCB)*, one for each file attribute that the caller is trying to read or write. In the figure, a process has opened the user-defined and data attributes. The SCBs representing all the attributes of a file point to a common data structure called a *File Control Block (FCB)*. The FCB, one per file, contains the file record number for the file in the MFT, i.e., the MFT entry number.

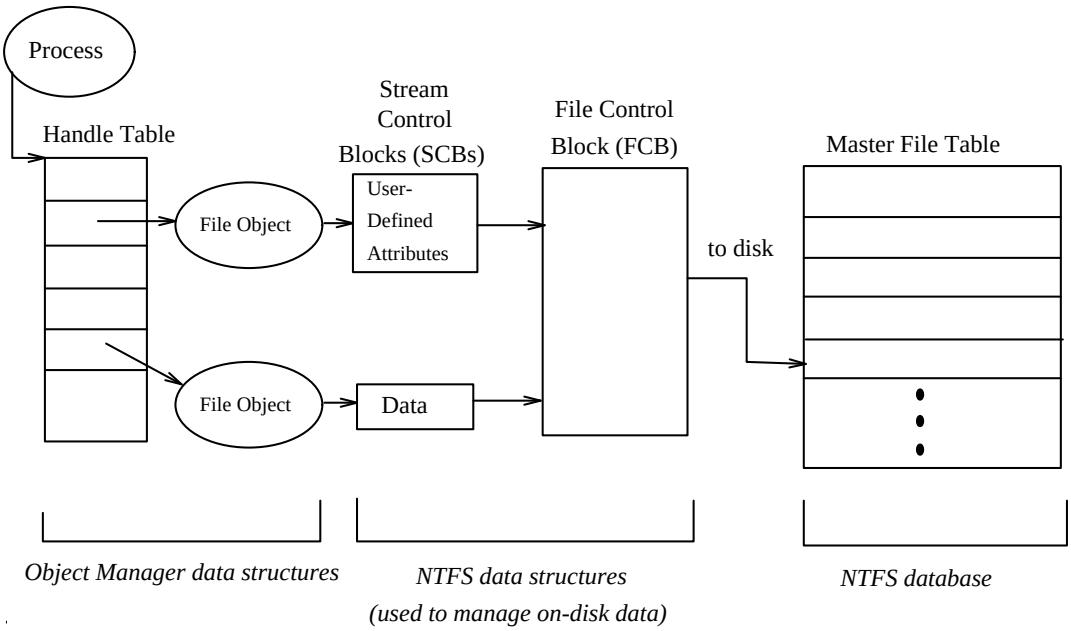


Figure 4.18. File Objects

NTFS is not supported on floppy disks.

4.1.7.3.3 The FAT File System The File Allocation Table (FAT) file system is the standard MS-DOS file system. A FAT file system is divided into two parts: a small system area used to store information about the disk, and the data area where file data is stored. The system area is separated into three parts: the *boot*, the *FAT*, and the *root directory*. A boot record may optionally be empty.

Windows NT supports FAT file systems on floppy disks and hard disks, but the evaluated configuration only supports hard disk FAT file systems on Alpha machines, where a FAT partition is needed in order to boot Windows NT.

Floppy disks are protected by the DACL associated with the device object representing the floppy device. When configured according to the TFM, the Winlogon server changes the DACL for the floppy device during the logon process so that only the current interactive user may access the device. Winlogon also ensures that any current handles to the device (e.g., from a process still active from a previous logon session) are invalidated.

For the bootable FAT partition required for the Alpha, the DACL for the device object can be changed via the Disk Manager administrative tool, allowing this partition to be protected.

4.1.7.3.4 The CDROM File System The CDROM file system supports both files and directories that are exported as file objects. Like with floppy devices, the CDROM files and directories are protected by the DACL associated with the CD device object. When configured according to the TFM, the Winlogon server changes the DACL for the CDROM device during the logon process so that only the current interactive user

may access the device. Winlogon also ensures that any current handles to the device (e.g., from a process still active from a previous logon session) are invalidated.

4.1.7.3.5 The Named Pipe File System (NPFS) As previously described, a named pipe is exported as a file object which represents an instance of the named pipe. A device object exists for NPFS providing a parse method that determines how the name of the named pipe is to be processed. The device object has a security descriptor with an access control list that indicates traverse access to Everyone and read/write access to administrators.

A named pipe provides a full duplex channel that can be used to implement an interprocess communication (IPC) mechanism between two processes. Named pipes provide the transport medium that is used by the Windows NT Remote Procedure Call (RPC) facility. The named pipe file system (NPFS) is another file system driver (FSD) within the Windows NT I/O subsystem.

Named pipes have two ends: a client end and a server end. Both ends of the pipe are full duplex, i.e., data written from one end can be read from the other end and vice versa. The server end of the pipe is created when a new instance of a named pipe is created or when a previously created instance is reused. The server end of a named pipe has a name and a DACL providing protection in much the manner as for NTFS files. Before either the client or the server end of a named pipe can be used, the server end must be connected. Once this is accomplished, the client end of the pipe can be created and opened and information can flow over the pipe. Named pipes are created with the following attributes:

- A type which is either *message* or *byte stream*
- A count that limits the maximum number of simultaneous instances of the named pipe that can be created
- An input buffer size that specifies the size of the buffer that is used for in-bound data on the server side of the named pipe
- An output buffer size that specifies the size of the buffer that is used for out-bound data from the server side of the named pipe
- A default timeout value that is used if a timeout value is not specified

The type of a named pipe determines how the information is written into the named pipe. If the named pipe is a message pipe, then information is written into the pipe in the form of messages which include the byte count and the data of the message. If the named pipe is a byte stream pipe, then only the data is written into the named pipe.

Named pipes can be in one of four states:

- Disconnected - the initial state; when the pipe is in this state, no client is connected to the pipe and a listen operation can be performed
- Listening - performing a listen operation on a disconnected named pipe causes the pipe to transition to this state
- Connected - an open request performed by a client causes a named pipe in the listening state to enter this state; data can then flow through the pipe
- Closing - A connected pipe can transition to either the disconnected state or the closing state

Key Field	Description
Name	Used to access key
TitleIndex	Index to human readable key name
Class	Kind of key, indicates how to interpret key
ACL	Access control list
LastWriteTime	Last time a modifying operation performed on key or value entries
Value Entry(s)	Contains name, type, and data field
Subkey(s)	Independent keys

Table 4.8. Key Object Structure

4.1.7.3.6 The Mailslot File System (MSFS) Similarly to named pipes, a mailslot is exported as a file object which represents an instance of the mailslot. A device object exists for MSFS providing a parse method that determines how the name of the mailslot is to be processed. The device object has a security descriptor with an access control list that indicates traverse access to Everyone and read/write access to administrators.

Mailslots are a form of interprocess communication (IPC). They provide a facility for unidirectional message passing. Each instance of a mailslot object has a DACL which affords protection in a manner similar to NTFS file objects. The mailslot file system (MSFS) is another file system driver (FSD) within the Windows NT I/O subsystem.

Mailslot messages consist of a message buffer and a priority. Priority zero message are written onto the end of the message buffer; higher priority messages are copied to the front of the circular message buffer if their priority is higher than the message at the head of the buffer.

4.1.8 Configuration Manager

The Configuration Manager is an executive subsystem which maintains a database and provides functions to manipulate the database entries. This database is referred to as the registry. The registry is used to store data of any kind by other parts of the system. It is mainly used as the central repository for configuration and control data.

Administrative tools such as the Registry Editor, Event Viewer, and the Control Panel interact with the registry to modify configuration data.

4.1.8.1 Key Object

The registry is a structured tree, with each node being a key object. A key can have child keys which are independent nodes in the tree called subkeys. Each key has an ACL controlling access to the key. The key must be opened with a specified access and a handle returned. The value and subkey fields are variable in length while the other fields are fixed. Registry data is maintained as value entries in the key object.

The key object structure is identified in Table 4.8.

4.1.8.2 Registry

The Configuration Manager manages its own name space below the pathname Registry in the global name space of the Object Manager (see Section "Global Name Space and Related Objects," on page 60). The registry is stored on disk. The value field in the key object is considered data (not interpreted by the registry). The key object is used to store various types of information in the system. Configuration Manager services are available to the executive and servers to store information in a key object. For example, the SAM server stores its internal objects as key objects in the registry (see Section "Security Accounts Database," on page 113).

By convention, the registry is organized into the following four subkeys:

- Key_Classes_Root
- Key_Current_User
- Key_User
- Key_Local_Machine

The Key_Classes_Root contains information about different filename extension associations and object linking and embedding (OLE). This subkey is the exact information contained in the Classes subkey under the Software subkey under the Key_Local_Machine subkey. The Key_Classes_Root subkey is provided here only for compatibility with the Windows 3.1 registration database.

The Key_Current_User subkey contains the currently logged on user's profile. A user profile defines the environment for a particular user (e.g., application preferences, screen colors). The information in this subkey takes precedence over system variables.

The Key_User subkey contains the actively loaded user profiles. This subkey has at least two subkeys: Default, and the currently logged on user's SID. The information in the Default key is used to create a user profile for a user who logs on without a profile defined.

The Key_Local_Machine represents the physical state of the computer containing information about the hardware and operating system data such as the bus type, system memory, device drivers, and start-up data. This subkey contains five subkey listed in below.

- Hardware - describes the physical hardware
- SAM - contains the Security Account Manager's database
- Security - contains the local security policy
- Software - contains data about software
- System - controls system startup and operating system behavior

The Hardware subkey contains the hardware data computed at system startup. This subkey contains information about hardware identification, driver information, mapping information of device drivers to resources it uses (I/O ports, I/O memory addresses, interrupts, etc.). The information in this subkey is volatile—it is computed each time the system is started and discarded at shut down. Data can also be added to this subkey after startup, as needed. For example, privileged users can load and start drivers which would set entries in this subkey. The SAM subkey contains the user and group account information in the Security Account Manager's (SAM) database. Domain information is included on the Windows NT Server.

The Security subkey contains security information such as user privileges, password policy, and group membership. There is also a SAM subkey under the Security subkey which is linked to the SAM subkey under Key_Local_Machine. This is done for historical and compatibility reasons.

The Software subkey contains specific information about software on the computer and configuration information associated with the software. This subkey also contains a subkey which is the same as the Key_Classes_Root subkey described earlier.

The System subkey contains all the start-up information needed to boot the system. To assist in ensuring that the system can be started, backup versions of the system are stored in this subkey. These backup versions are used to invalidate any configuration changes that did not perform as intended. This data is organized into subkeys called control sets and each control set has two parts: control data and services data. The control data part of a control set contains information used to control the system, such as the subsystems to start, computer-dependent environment variables, and the size and location of the paging files. The services part of a control set contains the services to be loaded and their order, such as the drivers. There can be as many as five control sets to represent a default control set number, current control set lastknowngood (LKG) control set, failed control set, and a clone control set. Although there can be as many as five, there is usually only two (the LKG and the default). These multiple control sets are used to resolve problems that may occur during start-up. The default control set will be initially tried the next time the system is booted. The current control set identifies the control set that actually booted the system. When a boot starts, the control set is copied to the clone control set before any changes can be made. When a boot is declared good, the clone control set is copied into a new control set, which becomes the LKG set and the old LKG is discarded. (The clone control set is discarded then also). When a boot fails, if the LKG is specified, then the LKG control set becomes the current (instead of default). If the LKG boots successfully, the LKG control set becomes the default, a new copy becomes the LKG control set, and the old default becomes the failed control set. This failed control set will be used for debugging and support. After setup, there should always be a LKG control set.

4.1.8.3 Hives

A hive is a subkey in the registry name space which is backed by a single file. Creating and manipulating a hive is a privileged operation requiring the Restore privilege. The following registry subkeys are hives: the SAM, Security, Software, and System under the Key_Local_Machine root, the Default subkey under Key_User root, and the Key_Current_User root. These hives are opened exclusively by the Configuration Manager (which runs under the context of the kernel) denying all read access to all others. Hives that are not in use (profiles for users not logged on) are not loaded or opened and are protected by their security descriptors.

4.1.9 Executive Object Services

Executive Object Services is responsible for exporting Microkernel objects outside the executive. The Microkernel only exports objects to the rest of the Windows NT executive. The Executive Object Services exports the following Microkernel objects outside the executive: mutant (called a mutex outside of the executive), event, event pair, semaphore, timer, profile, and queue. A queue object is exported as an I/O completion object; the entries in the queue list are I/O completion notifications. The Process Manager exports Microkernel thread and Microkernel process objects (see Section "Process Manager," on page 81).

The Executive Object Services subsystem also provides some internal functions to the rest of the Windows NT executive. None of these functions are exported outside of the executive.

4.1.10 Virtual DOS Machine

Windows NT supports the execution of 16-bit MS-DOS binaries by simulating a virtual DOS machine in a user-mode process. The MS-DOS program sees a combination of simulated and real hardware, and software, that looks just like a real-mode single-state PC compatible machine running MS-DOS. Real mode is the x86 processor mode used by DOS. The Windows NT Virtual DOS Machine (VDM) encapsulates DOS programs in an environment where they appear to have complete control of an x86 machine but are isolated from the Windows NT executive and all other processes.

VDM emulation is accomplished on the DEC AXP/150 by user-mode software that emulates the Intel x86 real-mode CPU instructions, the DOS operating system, the BIOS system calls, and the PC compatible machine environment. A different approach is taken on x86 machines because the x86 CPU has hardware support for simulating an x86 processor running in real mode while it is encapsulated within a user-mode process in protected mode. This mode obviates the software emulation of CPU instructions and results in higher performance. However, it is more complex than the software emulation of the AXP/150 and requires microkernel support.

When the program loader determines that an executable it has been requested to load is a DOS program it starts an NTVDM virtual x86 process and passes it the name of the DOS executable as an argument. NTVDM, a 32-bit user-mode program, requests 16 MB of virtual space, sets up a 16-bit DOS-like environment (including PC BIOS and DOS wedges) in this space, loads the DOS program identified in its argument, and calls the Windows NT executive function NtVdmControl (subfunction VdmStartExecution) to request a dispatch to the VDM. NtVdmControl sets the VM bit in the EFLAGS CPU register and dispatches to the process in Virtual x86 mode.

The BIOS and DOS emulators that NTVDM installs in its VDM environment call the executive to satisfy their requests for system services by issuing the BIOP instruction, which traps to the executive as an invalid opcode. The executive recognizes the VDM process and returns control to NTVDM by manipulating the stack so that the executive's IRET returns to NTVDM in 32-bit mode. NTVDM sees this as a return from its NtVdmControl (VdmStartExecution) call by which it caused the last dispatch to the VDM process. NTVDM satisfies the service request and calls NtVdmControl (VdmStartExecution) again to return control to the VDM process.

Attempts by the VDM process to access hardware, via the IN and OUT instructions, are likewise trapped, since IN and OUT are privileged instructions. After the trap, the stack is similarly manipulated to return control to NTVDM in user mode so it can emulate the hardware access and eventually return to the VDM process.

The ability for NTVDM to handle invalid instruction traps is unique and not provided to normal user-mode processes. It only works when the trap is generated by the VDM in Virtual x86 mode.

The DOS wedge in the x86 NTVDM and the Alpha DOS emulator do not manage FAT file systems. Rather, they convert DOS file system calls to Windows NT system calls. The results are returned back through the wedge as DOS call responses. Files on either NTFS (hard disk), CDFS (CD-ROM), or FAT (diskette) partitions (in the evaluated configuration) may be accessed by VDM programs. FAT file system names conform to the 8.3 format which DOS programs expect. NTFS can keep two names for each file, the NTFS

long name and a short (8.3 format) name. Unique short names are automatically generated for NTFS files which have long names, as the need arises (when they are accessed by DOS programs). Once generated, though, a short name endures until the file is renamed, replaced, or deleted. DOS programs see the long name, if it is 8.3 conforming, or else they see the short name.[19] Access to other system resources are similarly routed through Windows NT. The VDM cannot directly access hard disk drives or controllers.

When a DOS process is running in full screen mode, the video card's memory (but not its control registers) is mapped directly into the VDM's virtual address space. When a DOS process is running in a window, NTVDM creates a helper thread that periodically copies the contents of a virtual screen buffer in the VDM process to the window via Win32.

VdmStartExecution is one of eight subfunctions implemented by the x86 microkernel NtVdmControl executive call. All of them are:

VdmStartExecution: Switches to Virtual DOS Mode. Return is made from this routine when DOS invalid instructions (usually BIOP, IN, and OUT) are attempted.

VdmInitialize: Initializes a VDM process, reserving 16 MB memory starting at VDM address 0.

VdmQueueInterrupt: Sets up an interrupt routine in NTVDM to accept unsolicited interrupts from the executive. Used, for example, to emulate DOS timer tick interrupts.

VdmDelayInterrupt: Works with VdmQueueInterrupt to handle interrupts that cannot be processed immediately.

VdmQueryDir: Supports the DOS find-first and find-next file system functions.

VdmQueryDirFile: Supports the DOS find-first and find-next file system functions. These functions test a globbed expression (a pattern that contains wild card characters like * and ?) against the names of existing objects (files and directories) and returns those that match. Find-first specifies the glob pattern and returns the first file. Then find-next is called repeatedly to retrieve each subsequent matching name.

VdmSetInt21Handler: This is an optimization that allows NTVDM to trap the DOS INT 21 system call more efficiently.

VdmFeatures: Returns machine and configuration information.[20]

4.2 Protected Servers

TCB protected servers run in a user-mode TCB process with a security context of SYSTEM. A security context of SYSTEM means that the protected server's token is defined as shown in Table 4.9.

The protected servers in the evaluated configuration include: the Session Manager; the Service Controller; Win32; WinLogon; the Local Security Authority (LSA); the Secure Account Manager (SAM); the Event

[19] The Windows NT command shell's DIR command has a new /x command line switch which displays both the long and 8.3 format names of each selected file in an NTFS partition.

[20] There is Pentium-specific code in Windows NT that is used whenever the presence of a Pentium CPU is detected. It is currently disabled by forcing the CPU model detection routine to report a 486 when it detects a Pentium.

Field	SYSTEM Token Value	
User ID	SYSTEM	
Group ID array	Everyone Administrators	
Owner ID	Points to Administrator group ID.	
Privilege(s)	TCB (enabled) TakeOwnership (disabled) LockMemory (enabled) IncreaseQuota (disabled) CreatePermanent (enabled) Audit (enabled) SystemEnvironment (disabled) Backup (disabled) Shutdown (disabled) ProfileSingleProcess (enabled)	CreateToken (disabled) CreatePagefile (enabled) AssignPrimaryToken (disabled) IncreaseBasePriority (enabled) Debug (enabled) Security (disabled) ChangeNotify (enabled) Restore (disabled) LoadDriver (disabled) Systemtime (disabled)
Default DACL	SYSTEM GENERIC_ALL Everyone GENERIC_EXECUTE	
Source	Not used for SYSTEM token.	
Type	Primary	

Table 4.9. Token for Processes Running with the SYSTEM Security Context

Logger; and the Print Spooler. Each protected server provides a set of functions available to other protected servers as well as untrusted clients. Access to these functions is via a client-server model using some form of IPC. All protected servers, with the exception of Win32, use the RPC protocol via an LPC port to communicate with clients. The IPC mechanism used by Win32 is discussed in Section "Win32," on page 118. Each of the protected servers are discussed in this section.

4.2.1 Session Manager

The Session Manager is started by the system software initialization program. Its primary function is to initialize DLLs and load the DOS device drivers and subsystems listed in the Registry. Subsystems include the Win32 protected server and are listed as either optional or required. If required, the subsystem must be loaded by the Session Manager during bootup. In the evaluated configuration only Win32 is required.

The Session Manager also starts up Winlogon.

Content-Length: 19414

4.2.2 Security Subsystem

Figure 4.19 depicts some of the major security components of Windows NT. The Local Security Authority (LSA) is the central security component on a particular machine and maintains information such as which users and groups can logon to the machine, the privileges that are assigned to users and groups, and security events and parameters. The LSA is described in detail in Section "Local Security Authority," on page 109.

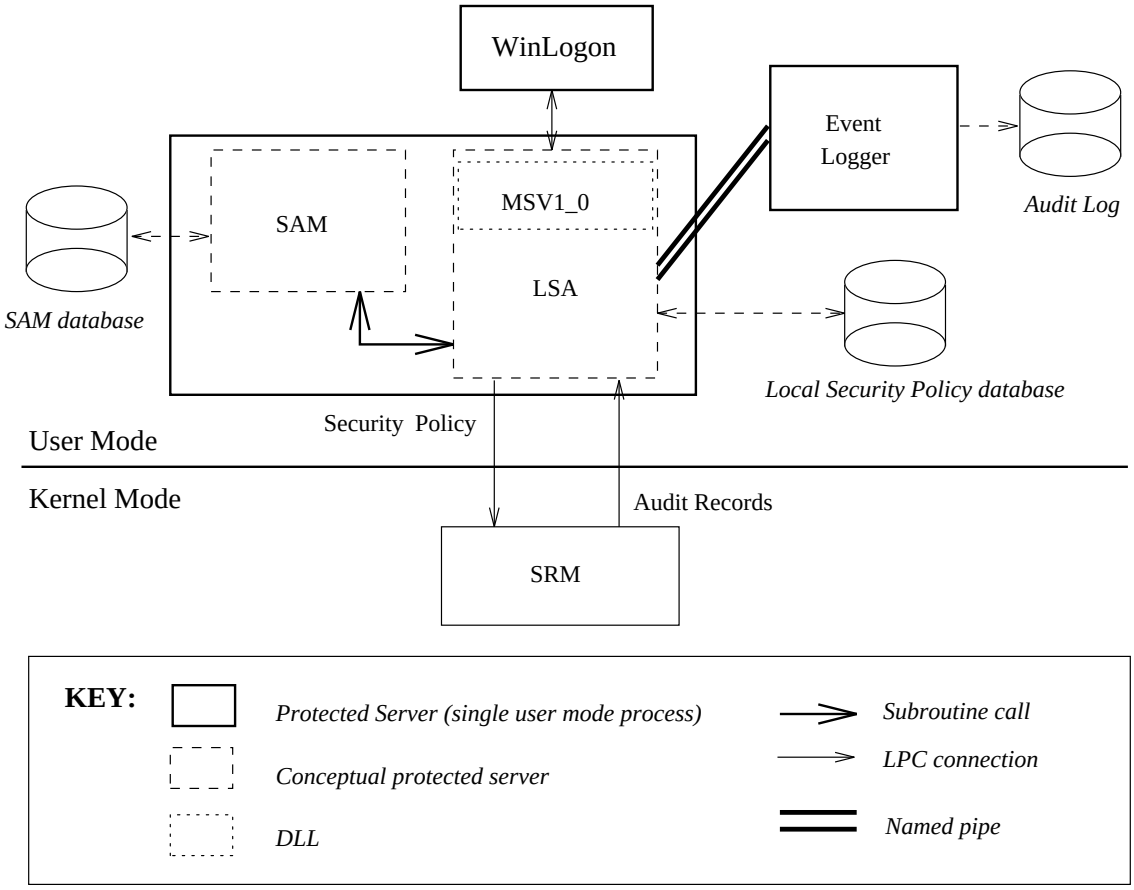


Figure 4.19. Major Security Components of Windows NT

The Security Accounts Manager (SAM) is responsible for managing the user accounts database on a particular machine. In addition, if a domain is defined, a SAM may be a domain server and maintain the user accounts database for all machines in a domain. A *domain* is defined as a group of machines where one machine is an Windows NT Server and acts as the domain server. Only an Windows NT Server can be a SAM domain server. In the evaluated configuration a domain of one machine can be defined. The SAM is discussed in Section "Security Accounts Manager," on page 112.

MSV1_0 is an authentication package which performs the actual user authentication. This package is discussed in more detail in Section "LSA and Authentication," on page 111.

The LSA communicates with the Event Logger to record security event records. The role of the LSA in security event record generation and writing the security event record to the audit log (via the Event Logger) is described in Section "Audit Mechanism," on page 158 (see Section "Startup Audit Records," on page 163 and Section "Construction of Audit Records," on page 163 in particular).

The LSA communicates with the SRM in order to send the local security policy to the SRM. This is discussed in detail in Section "LSA and Authentication," on page 111.

Conceptually, LSA and SAM are thought of as separate protected servers, however, they are actually implemented as a single process. LSA and SAM use a set of private interfaces exclusively for their use via subroutine calls. MSV1_0 is implemented as a DLL which allows the LSA to link to several authentication packages. However, only MSV1_0 is included in the evaluated configuration. Communication between the LSA and WinLogon, and between the LSA and the SRM, is via the LPC facility (see Section "Local Procedure Call Facility," on page 82 for a discussion of LPC). Communication between the LSA and the Event Logger is via a named pipe.

4.2.2.1 Local Security Authority

The LSA has three main functions:

- Management of the local security policy database
- Participation in the authentication process
- Participation in the security event logging process

The local security policy database is discussed in detail in Section "The Local Security Policy Database," on page 110; the interaction between the components depicted in Figure 4.19 during the authentication process is discussed in Section "LSA and Authentication," on page 111; and the role of the LSA in security auditing is described in Section "LSA and Audit," on page 112.

4.2.2.1.1 LSA Initialization The security subsystem protected server (which includes both LSA and SAM) is created by WinLogon during the software initialization phase of the boot process. WinLogon is a *logon process*. A logon process is a process that is assigned to monitor a device for logon attempts and has registered as a logon process with the LSA (see Section "WinLogon," on page 115 for more details about WinLogon). In order to successfully complete the registration process, WinLogon must reside on the same machine as the LSA and must possess the TCB privilege. To register, WinLogon calls the LSA via RPC. The LSA impersonates WinLogon and checks for the TCB privilege before setting up the communication ports for an LPC connection.

The LSA manages the local security policy database. Like all protected servers, the LSA executes with the SYSTEM token. The local security policy database is protected by a DACL which allows for all access to SYSTEM.

As part of its initialization, the LSA examines its local security policy database for the list of authentication packages available on this machine. In the evaluated configuration, the only authentication package is MSV1_0, which is used by WinLogon. The LSA links to MSV1_0, which is a DLL, and calls an initialization routine that allows MSV1_0 to initialize itself. The LSA sends a dispatch table to MSV1_0 which contains entry points for a set of routines provided by the LSA for use by MSV1_0. For example, most of the storage used by MSV1_0 is allocated from the LSA's heap. The dispatch table contains entry points that allow MSV1_0 to allocate/deallocate required storage. The LSA also maintains a private data structure for storing credentials that may be created by authentication packages. The dispatch table contains entry points for adding, retrieving, and deleting credentials. The LSA does not interpret the credential information; the format and meaning of the credentials is specific to an authentication package. The credentials created by MSV1_0 are discussed in Section "LSA and Authentication," on page 111.

During initialization, the LSA connects to a port created by the SRM specifically for the LSA; SRM, in turn, connects to a port create by the LSA for communication from the SRM to the LSA. The LSA then sends the SRM the local security policy via the LPC connection. Subsequently, the LSA uses this connection to notify the SRM of updates in the security policy, and creation/deletion of a logon session. Several shared memory sections are also created by the SRM for exchanging large messages that can't be passed over an LPC connection. This is described more fully in Section "Audit Policy," on page 79. The port created by the LSA for the SRM will be used by the SRM to send audit records to the LSA.

Finally, the LSA creates a named pipe for subsequent communication with the Event Logger. The LSA uses the services provided by the Event Logger to record audit records in the audit log (see Section "Event Logger and Audit," on page 159).

4.2.2.1.2 The Local Security Policy Database The local security policy database contains the following machine-specific information:

- Domains trusted to authenticate logon attempts[21]
- Users and groups that may access the system and the manner in which they may access it (the type of logon allowed, which is described in the next section)
- Privileges assigned to users and groups
- Security events and parameters
- Default memory quotas for users

The local security policy database is stored in the registry and is accessed directly only by the LSA. The local security policy database may contain subobjects representing accounts and secrets.[22] Access to these subobjects is determined by the DACL on the subobject.

The subobjects representing accounts represent a single user or group. For each user or group, the subobject also lists the set of privileges to be assigned to the user or group, the type of access that the user or group is given to the system (interactive, network, or service - described in the next section), and a quota assignment.

The subobjects representing secrets is used to store secrets such as the password for a service logon. However, in the evaluated configuration, none of the services installed by the Service Controller require a password. Therefore, at installation, there are no such subobjects present. If such a service is subsequently started, the appropriate information will be added to the local security policy database. (see Section "Service Controller," on page 116).

The local security policy object also maintains information about the audit log.

- The percentage the audit is full
- The maximum size of the audit log
- The retention period
- A flag which indicates whether a shutdown is in progress because the audit log has become full
- The time of that shutdown.

[21] Only in the networked environment and not in the evaluated configuration.
[22] The local security policy database can also information about domains, but in the evaluated configuration, it does not.

In addition, the events for which auditing is enabled and disabled are also recorded. After installation, an administrator uses the User Manager interface (see Section "User Manager," on page 131) to maintain the local security policy.

4.2.2.1.3 LSA and Authentication The LSA supports three types of logons: *interactive, service*, and *network*. In the evaluated configuration, there is one logon process that handles interactive logons by monitoring the WindowStation for the secure access sequence (SAS)—WinLogon (see Section "WinLogon," on page 115).[23] There is also one logon process that handles service logons, the Service Controller. However, in the evaluated configuration, none of the services that are started by the Service Controller during installation require logon (see Section "Service Controller," on page 116). Network logons are not supported in the evaluated configuration.

For an interactive logon, Win32 collects the password from the password dialog box displayed and uses a simple encryption algorithm known only to Win32 to immediately encrypt the password. Win32 then temporarily decrypts the password in order to pass it along to WinLogon (see Section "Win32," on page 118 for more details about Win32) where it is immediately encrypted by WinLogon using a simple encryption algorithm known only to WinLogon and MSV1_0 (see Section "WinLogon," on page 115 for more information about WinLogon). The purpose of temporarily encrypting the password in both places is to minimize the possibility that a clear text password will find its way to a paging file.[24] In addition to the password, WinLogon also collects the username and domain. Lastly, WinLogon creates a unique *logon SID* for the user that will be assigned to the token of the new process as well as the security descriptor of this instance of the WindowStation, after a successful authentication. The logon SID is not stored in the security accounts database and is destroyed after logoff. WinLogon packages the username, encrypted password, domain, the logon SID, name of the authentication package (in this case MSV1_0), and subsequently calls the LSA.

The LSA doesn't know how to interpret this information so it calls MSV1_0, the authentication package, passing along the information it received from WinLogon. MSV1_0 does know how to interpret the information, and first establishes the type of logon being requested, in this case an interactive logon. Given that, MSV1_0 can then interpret the rest of the information and retrieves the username, encrypted password and the domain.

MSV1_0 next establishes the credentials which it will subsequently use as part of the authentication process. The credential contains a single primary key which includes three pieces of information: the username, the hashed clear text password, and the case-sensitive hashed password. MSV1_0 does not decrypt the password until it is ready to run the clear text password through its one-way-hash function. The hashed password is what is ultimately needed for authentication (see Section "Support for Authentication," on page 113 for more information about how SAM stores passwords). MSV1_0 stores the credentials in the local variable provided by the LSA using the credential entry points identified in the dispatch table.

MSV1_0 now accesses the security accounts database via SAM (all access to the security accounts database is via SAM as described in Section "Security Accounts Manager," on page 112) to retrieve the hashed password for this user and compares the two values. If the password is valid, MSV1_0 retrieves from the security accounts database the user SID and set of group SIDs assigned to the user account. MSV1_0 also retrieves profile information (such as the logon time, date the password was last changed and so on) from the security accounts database. MSV1_0 then allocates space and stores this information in the LSA's heap

[23] Win32 actually manages keyboard input and notifies WinLogon that the SAS has occurred. See Section "Win32," on page 118 for more details.

[24] Paging files are protected by an exclusive lock, acquired during initialization, so that no one can read the paging file.

and returns a pointer. Before returning to the LSA, MSV1_0 also allocates the *authentication ID* (also called the logon ID)[25] which is used by the LSA and the SRM to associate all instances of tokens for a particular logon session.

The LSA subsequently incorporates the local security policy by performing a filter/augmentation step. As part of the filter/augmentation step, the LSA accesses the local security policy database to collect any assigned local groups and/or assigned privileges. The LSA augments the list of group SIDs with well-known local group SIDs. Examples of well-known local groups are Everyone, which is the group that includes all users; interactive, which is assigned to users logged on interactively; and local, which is assigned to users logged on to terminals physically connected to the machine. The LSA then scans the list of SIDs to determine if the user has the right to logon interactively. If not, the LSA notifies MSV1_0 that the logon has failed.

If the logon is successful, the LSA adds the logon SID (created by WinLogon as previously described) to the list of local groups. The LSA then fabricates the default ACL such that the owner SID and SYSTEM have all access. Most of the time the owner SID is equivalent to the user SID, however, if the user is a member of the Administrators group, the owner SID will be Administrators rather than the user SID.

The LSA then creates the primary token by calling the executive. The LSA is the only place in the TCB where a primary token is created. The LSA also passes the authentication ID to the SRM so that the SRM can maintain the token count for the associated logon session.

Once the new token is successfully created, the LSA duplicates a handle to the token and returns it to WinLogon. WinLogon previously created a new user process when it received the logon request and suspended it pending user authentication. The PCB in the suspended process points to a primary token that is an exact copy of its parent, in this case WinLogon. Once WinLogon receives the handle to the new primary token from LSA it calls the Process Manager to replace the primary token in the PCB with the new token, and unsuspends the process.

Subsequently, if the token count in the SRM goes to zero, all threads in all processes in the session have exited and the SRM notifies the LSA that the session is terminating. The LSA calls MSV1_0 so that the authentication package can perform cleanup activities such as deleting the credentials associated with the logon session. Alternatively, the LSA can notify the SRM to terminate the logon session. WinLogon destroys the logon SID at logoff.

4.2.2.1.4 LSA and Audit The LSA provides parameters to the SRM which dictate the event types for which the SRM will generate audit records. This information is sent from the LSA to the SRM during initialization as part of the local security policy as described in Section "LSA Initialization," on page 109. Updates to the local security policy are also sent to the SRM. The role of the LSA in audit generation and writing the audit record to the audit log (via the Event Logger) is described in Section "Audit Mechanism," on page 158 (see Section "Startup Audit Records," on page 163 and Section "Construction of Audit Records," on page 163 in particular).

4.2.2.2 Security Accounts Manager

The Security Accounts Manager (SAM) is designed to support an administrative security realm that may extend beyond a single machine, called a *security domain*. The SAM is responsible for managing the security

[25] The authentication ID is not a SID; it is also called a logon ID but should not be confused with the logon SID created by WinLogon. The authentication ID is an internal counter and it is unique.

accounts database which contains user and group account information for a particular machine. If the SAM is a SAM domain server, as previously described, the SAM is also responsible for domain-wide security information. In the evaluated configuration, however, only a security domain of a single machine may be defined.

All access to the security accounts is via the SAM. As will be described throughout this section, access to SAM objects is controlled via the DAC mechanism.

The SAM protected subsystem has two primary functions:

1. Maintenance of user and group accounts
2. Support for authentication packages

Administrators use the User Manager interface (see Section "User Manager," on page 131) to create and maintain the security accounts. SAM provides security account administration services via RPC. Interaction with the authentication packages, in this case MSV1_0, is via subroutine call.[26]

4.2.2.2.1 Security Accounts Database The security accounts database is stored in the registry and is accessed directly only by the SAM. The security accounts database contains subobjects representing users, groups, aliases, and any defined domains.[27] Access to these subobjects is determined by the DACL on the subobject.

The subobjects representing users contains information about an individual user, such as user preferences, logon information, and account information. Table 4.10 lists some of the information stored in the secure accounts database with respect to users.

The subobjects representing groups contains a list of user SIDs that are members of the group as well as attribute information about those members and the group as a whole. Some of the information contained in the secure accounts database with respect to groups is shown in Table 4.11. The subobjects representing aliases are the same as those representing groups with the exception that member SIDs can be either user SIDs or group SIDs. An alias is also called a local group.

4.2.2.2.2 Support for Authentication As previously described, administrators use the User Manager interface to create security accounts and the SAM subsequently stores the account information in the security accounts database. During security accounts administration, the User Manager communicates directly with the SAM via the RPC services provided by the SAM.

Of particular interest is the storage of account passwords. The User Manager uses a one-way function to hash the password and then temporarily encrypts the hashed password, using a trivial session key, for transmission to the SAM. When the SAM receives the password, it decrypts it, again using the trivial session key, and re-encrypts it using a private key for storage in the user's account in the security accounts database. The private key that SAM uses is based on the account's user SID.

[26] In a networked environment, the authentication package is distributed. Networks are not supported in the evaluated configuration.

[27] Two Domain subobjects are created at initialization time for both the Windows NT Server and Windows NT Workstation products. This is an implementation detail done for symmetry of code.

Information Type	Purpose
Account Expiration	Time and date when account will expire; and expired account is automatically disabled (see User Account Control below).
Administrator Comment	A string containing an administrative remark or comment.
Bad Password Count	Number of attempts since the last successful logon where the authentication process resulted in a bad password.
Full Name	Full name of the user
Home Directory and Drive	User's home directory and drive on which it is located; may be NULL.
Last Logoff/Logon	Time and date when the user last logged off/on.
Logon Count	Number of times the user is currently logged in.
Logon Hours	Hours when the user is allowed to logon.
Password	User's password stored in hashed, then encrypted form.
Password Last Set	Time the password was last changed.
Password Changeability	Two flags that indicates whether the password can change or must change.
Primary Group SID	The relative SID of the primary group to assign or assigned to the user. For assign operations, this group must be one of which the user is a member.
Profile and Script Path	Pathnames of the user's windows profile script and logon script.
User Account Control	A bitmap used to identify account status such as account disabled, home directory required, password not required.
User Comment	A user-controlled field for comments to other users; part of the general information that can be retrieved about a user.
User SID	Relative SID of the user.
User Name	Name of the user as well as the account name.
Work Stations	List of workstations from which the user is permitted to logon; NULL means all workstations.

Table 4.10. User Information

Information Type	Purpose
Administrator Comment	A string which may contain administrator information about the group.
Attributes	Attributes of the group as a whole including whether the group can be disabled (which would cause the group to be ignored during access validation), whether the group is disabled by default, and whether the group may be assigned as the owner of objects. Attributes can also be set for individual members and cannot conflict with those set for the group as a whole.
Group SID	Relative SID of the group.
Member Count	Number of members of the group.
Member SIDs	List of user SIDs that are a member of the group.
Name	Name of the group.

Table 4.11. Group Information

The main role of the SAM during the authentication process is to provide requested information to the LSA and the authentication package, MSV1_0. Again, of particular interest is the user's password. When MSV1_0 requests the password from the SAM, the SAM first decrypts the password, using its private key, and returns the hashed value to MSV1_0. Thus, the authentication package is comparing the two hashed values, rather than clear text, or encrypted values.[28]

4.2.3 WinLogon

WinLogon is the "logon process" for Windows NT, and is started by the Session Manager. These are its functions:

- Coordinate the interactive logon and logoff process.
- Manage the Desktops.
- Provide the user interface for the interactive logon and logoff operations.

In order to ensure that it can maintain control of the WindowStation once the system is up, WinLogon registers itself with Win32 during system initialization as the logon process; attaches the keyboard, mouse, and monitor to the WindowStation created by Win32; assigns a security descriptor to the WindowStation, containing only one ACE for the WinLogon SID (thereby assuring that no other user can get access to the WindowStation unless allowed by WinLogon); and creates and opens a WinLogon Desktop. The result of this is that all secure operations[29] are performed only in the WinLogon Desktop. WinLogon then establishes an LPC connection with LSA, registers itself there as the logon process, and makes the WinLogon Desktop active.

WinLogon is the only process which can intercept logon requests from the keyboard (because it is first to register with Win32 as a logon process). After Windows NT boots, Win32 displays the "Welcome" screen, which welcomes the user and provides instructions to press the Secure Attention Sequence (SAS) (Control-Alt-Delete) to logon to the system. When the SAS is entered, WinLogon is called to initiate the identification and authentication process. WinLogon then brings up its secure Desktop and prompts for the username and password. When the username and password are both entered, WinLogon calls LSA, passing it the logon information and the name of the authentication package which it wants to process the logon information (see Section "Local Security Authority," on page 109).

If the authentication fails, WinLogon receives an appropriate message from LSA, which it displays on the WinLogon Desktop, and then prompts for new logon information.

If the authentication succeeds, WinLogon receives a handle to the token created for the user, which it uses to add an ACE for the user to the DACL on the WindowStation. WinLogon creates an application Desktop and a screensaver Desktop. It changes the DACLs of these desktops so that both the user and Winlogon has access to them.

After the user logons, pressing the SAS will activate the Winlogon desktop. The user then has six choices on a dialogue box: Lock Workstation, Change Password, Logoff, Task List (to switch tasks), Shutdown (with option to reboot), and Cancel, which returns the user to the application desktop.

[28] The hash algorithm is the Secure Hash Algorithm developed by the National Institute of Standards and Technology as published in FIPS PUB 180 (1993).

[29] Secure Operations refer to logging on and off, changing the password, shutting down the system (with optional reboot), and locking and unlocking the desktop. These are all operations that the user can choose after pressing Ctrl-Alt-Delete.

Locking the workstation switches control to Winlogon. A "Workstation Locked" message box appears stating that only the user or an administrator can unlock the workstation. This require pressing the SAS and entering the user's password or changing the username to an administrator and entering his password.

While the workstation is locked, any applications that were running continue running but cannot write to the screen. Output for the physical display will be written to a virtual display. If an administrator different from the interactive user who locked the workstation unlocks the workstation, the interactive user's applications will terminate. The administrator will not be able to determine what applications the user was running.

WinLogon also plays an important role in logoff. For information on logoff, refer to Section "Initialization and Shutdown," on page 131.

4.2.4 Service Controller

The Service Controller, also called Service Control Manager, is started by WinLogon during bootup (see Section "Initialization and Shutdown," on page 131). Some of the important functions of the Service Controller are:

- Maintain a database of installed services and the status information for each service.
- Start some services automatically at boot and others on demand.
- Track changes to services defined in the service portion of the configuration registry
- Manage services and reporting service status information.

Only services defined in the configuration registry can be run by the Service Controller (see Section "Configuration Manager," on page 102). An installed service is represented by a service object. An instance of this object type is created when the service installed and is deleted when the service is removed. This service object resides within a database of installed services managed by the Service Controller. It should be noted that no more than one instance of a service can be running at a time.

The Service Controller enables the following privileges: Backup, Restore, Security TakeOwnership, SystemEnvironment, Shutdown, AssignPrimaryToken, IncreaseQuota and LoadDriver.

Windows NT supports a special type of logon called *service logon*. This is the type of logon used by the Service Controller to activate a service defined in the registry. In order to do this, the Service Controller plays a similar role for the service logon as the WinLogon does for the interactive logon. In other words, to run any installed service, the service has to have a user account and an associated password. This identification has to be authenticated using the authentication services provided by LSA (see Section "Local Security Authority," on page 109), and SAM (see Section "Security Accounts Manager," on page 112). In order to do this, the Service Controller registers itself with the LSA as a logon process. (Note that the Service Controller has the necessary TCB privilege to register with LSA as a logon process). The authentication is done on the user account and password in exactly the same fashion as done for interactive logon. If the authentication fails, the corresponding service will not be allowed to run.

By default, a service runs in the SYSTEM account and hence requires no password. In this case, no call is made to LSA and authentication is not required. The service is executed as a thread within the Service Controller's process.

In the evaluated configuration of Windows NT, the only service that is included in the registry is the Event Logger service. This service runs in the default SYSTEM context. There are no services in the evaluated configuration that run in a user account.

4.2.5 Event Logger

The Event Logger provides a means to collect and store system or application events. It is the first service to be started by the Service Controller during the system bootup. The Event Logger records events relating to an application, the system, or security to an event log. These event logs can be informational notes, warning messages, error messages, or security audit records. The Event Logger exports the log object. There are three instances of this object: application log (used for application specific events), system log (where the system logs non-security relevant events), and the security log (where the system logs security relevant events). Access to all three instances of the log object is controlled by the DACL on these objects. However, access to the security log file is additionally controlled by the Security privilege (see Section "DAC Related Privileges," on page 146). Only a member of the Administrators group has access to the security log.

An event log header contains the fields as described in the Table 4.12. Each event log will contain the event header followed by event-type specific data.

Field	Description
Length	Length of this log entry.
Reserved	Reserved for future use.
RecordNumber	Contains a record number to begin reading at a specified number.
TimeGenerated	The time when this entry was submitted by the event generator.
TimeWritten	The time when this entry was received by the Event Logger Service.
EventID	Identifies the actual event. To be used with the module name.
EventType	The type of event: informational, error or warning.
Numstrings	The number of strings present in the log.
EventCategory	A subcategory of the event (module specific).
ReservedFlags	Reserved for use by Eventlog service.
ClosingRecordNumber	Closing Record Number.
StringOffset	The offset of the strings within this event log entry.
UserSIDLength	The length of bytes of the UserSID field.
UserSIDOffset	The offset of the SID within this event log.
DataLength	The length of the event-specific data in bytes.
DataOffset	The offset of the event-specific data within this log.
ModuleName	The variable length name of the module.
ComputerName	The name of the computer that generated this event.
UserSID	The SID of the active user when this event was logged.

Table 4.12. Fields of an Event Header

The system administrator manages the event logs by specifying the event logging management information such as the fully qualified path name for the file, the number of categories of events that can be supported, the

retention period, and the maximum size. This information is stored in the services section of the configuration registry.

Each record in the event log may be kept for a retention period. The default is seven days. The log is used as a circular buffer: the records wrap around and overwrite the existing records as long as the record to be overwritten does not fall within the retention period. However, the system can be configured to not overwrite events. When the log size equals its maximum, the system will display a warning that the log is full, and, if the CrashOnAuditFail key in the registry is set, the system will shutdown.

The entries in the Event Logger's security log can be viewed by the "Event Viewer" (see Section "Administrator Tools," on page 128).

4.2.6 Win32

The Win32 protected server[30] is a process that provides the principal user interface to Windows NT. To the user, this interface is the combination of a screen, keyboard, and mouse, represented by a *WindowStation* object; and a view of objects on the screen, called a *Desktop* object. These are the only two objects exported by Win32. Win32 objects are discussed in detail in Section "Win32 Objects," on page 120.

Win32 clients are processes that request services from the Win32 server. Although the Windows NT executive creates processes (see Section "Process Manager," on page 81), the Win32 server maintains additional information about all client processes and threads. This information is stored in a process and thread data structure in Win32 memory. Clients communicate with Win32 via the LPC (see Section "Local Procedure Call Facility," on page 82 for more information about LPC) or the Quick LPC message passing facilities, discussed later in this section.

Like all protected servers in the evaluated configuration, Win32 runs with the SYSTEM token as its security context. In addition, Win32 uses three privileges: IncreaseBasePriority, Shutdown, and Audit (see Section "Privileges," on page 151 for more details about privileges). The IncreaseBasePriority privilege is needed to test whether a process can be created with realtime priority; the Process Manager checks whether Win32 has this privilege before allowing Win32 to create such a process. The Shutdown privilege is needed when a critical error occurs within Win32 allowing orderly termination of Win32 within the executive. The Microkernel checks that Win32 has this privilege before allowing Win32 to cause a shutdown. Win32 uses the Audit privilege to request the generation of an audit record. This privilege is checked by the SRM.

4.2.6.1 Win32 Initialization

The Win32 server is bootstrapped during initialization of Windows NT. The initialization of Win32 includes the following steps, which are described in this section:

- Creation of the Win32 global handle table,
- Registration of Win32's logon process,
- Creation and initialization of a WindowStationobjects, and
- Creation and initialization of a Desktopobjects.

[30] Henceforth, the term Win32 when used alone is used to refer to the Win32 server.

Win32 maintains a *handle table* in its shared memory that contains information about Win32 objects and object attributes. The handle table can be seen, but not modified, by all Win32 client processes. Win32 objects, object attributes, and the handle table are discussed in detail in Section "Win32 Objects," on page 120. Initially, the handle table is empty.

WinLogon calls Win32 to register itself as Win32's logon process. Since only one process is allowed to register with Win32 as its logon process, Win32 first checks to make sure that a logon process has not already been registered. Win32 impersonates WinLogon and checks that the requesting process has the TCB privilege.[31] If successful, WinLogon is registered as Win32's logon process and WinLogon's process ID (PID) is stored in a local variable in Win32 memory for future verifications. Also as part of initialization, WinLogon registers with Win32 to receive notification of the Secure Access Sequence (SAS), which is the Ctrl-Alt-Del keyboard sequence. This is accomplished via the *hook* mechanism provided by Win32 which allows a process to register to receive particular keyboard sequences (all keyboard input goes to Win32). Hooks are described in detail in Section "Hooks, Cursors, and Icons," on page 121. The SAS is how the user requests a logon.

As previously described, the WindowStation assigned to the interactive logon provides access to the screen, keyboard, mouse, and other attributes that will be described later in this section (see Figure 4.20). This interactive WindowStation is created by WinLogon during initialization. Other WindowStations objects, which do not provide access to the screen, keyboard, and mouse, are created by Win32 on demand to support Win32 operations performed by services (see Section "Service Controller," on page 116 for a discussion of services).[32] A service logon, which has an associated account, can only have one WindowStation. See Section "LSA and Authentication," on page 111 for more information about the types of logons. If the creation of the WindowStation is successful, Win32 allocates storage for the WindowStation and initializes the fields of the WindowStation data structure which is inserted on the *global WindowStation list* in Win32 memory. The global WindowStation list is a linked list of WindowStation objects.

There are three predefined instances of Desktop objects for interactive logons: WinLogon, ScreenSaver, and Applications. The WinLogon Desktop and the Applications Desktop are both created by WinLogon and are sharable. The ScreenSaver Desktop is created on demand. WinLogon creates the WinLogon Desktop during initialization and it persists across all interactive logons. The Applications Desktop is created by WinLogon when an interactive user logs on and it is deallocated when the user logs off. The Applications Desktop is the only Desktop accessible to user processes. The ScreenSaver Desktop is created when a screen saver process starts and is destroyed when the screen saver process terminates. Other Desktops can be created on demand to support Win32 operations performed by services, provided the process has the access required by the parent WindowStation to create a Desktop. When a Desktop object is created, a Desktop data structure is allocated in Win32 memory and inserted in the *global Desktop list*, pointed to by the WindowStation data structure. The global Desktop list is a linked list of Desktop objects. Win32 allocates shared memory for each new Desktop and maps the shared memory into the address space of the WinLogon process. The WinLogon process subsequently attaches to the Desktop shared memory.

At this point, the handle table, which was previously empty, contains one entry for each WindowStation and Desktop that has been created, in this case the WindowStation that represents the screen, keyboard, and mouse; the WinLogon Desktop; and the Applications Desktop. Each handle table entry points to a security descriptor. At system initialization (prior to any user logging on), the security descriptor has the following relevant values:

[31] The TCB privilege is required of any process that wants to call itself a *logon process*.

[32] Non-interactive WindowStations have all the WindowStation attributes but no screen, keyboard, and mouse, and any attempt to manipulate the screen, keyboard, and mouse will fail.

- User SID = SYSTEM
- Group SID = SYSTEM
- Owner SID = SYSTEM
- DACL
 - SYSTEM - GENERIC_ALL
 - Administrators - ReadAttributes, Enumerate

When the user requests a logon by invoking the SAS, logon begins. The interaction between Win32, WinLogon, LSA, MSV1_0 (the authentication package), SAM, and the SRM, to establish an interactive logon is described in detail in Section "LSA and Authentication," on page 111 and Section "WinLogon," on page 115.

One of the first steps WinLogon does, prior to calling the LSA, is to create a new process for the interactive logon and suspend that process. When LSA returns successfully to WinLogon, a new token is returned which WinLogon substitutes for the primary token in the suspended process. The new primary token includes a logon SID that was fabricated by WinLogon during the early phases of the logon process. WinLogon provides this logon SID to Win32 who sets the User SID, Group SID, and Owner SID in the WindowStation and Desktop security descriptors to the logon SID. Win32 also changes the DACL so that the User SID also has GENERIC_ALL. In this manner, Win32 restricts access to the WindowStation and Desktop objects to the logged on user. The WinLogon process switches the user to the Applications Desktop from which all Win32 clients are subsequently started, and unsuspends the newly created process. Only the WinLogon process can switch Desktops. When the client starts up, the handle table in Win32 shared memory gets mapped read-only into the client's address space. The Applications Desktop shared memory gets mapped read-write into the client's address space so that the client can access Desktop attributes, such as Windows and Menus. If the client has the handle to the Applications Desktop, the client can attach to the Desktop shared memory.

When a client process issues a logoff, the Win32 server examines its list of threads to determine which ones belong to the authentication ID that issued the logoff. The authentication ID was created during logon by the authentication package and is used to keep track of all tokens for the same logon session. The authentication ID is explained more fully in Section "LSA and Authentication," on page 111. All such threads are sent notification to shut down. After a period of time, Win32 will automatically terminate any such threads that are still running. Win32 destroys communication with all other threads, except any that are running with the SYSTEM token. The security descriptors for the interactive WindowStation, WinLogon and Applications Desktops is reset to the initialization state (as previously described). Thus, no user-mode threads can access the WinLogon and Applications Desktops. Win32 marks handles for Win32 objects closed; handles for object attributes are marked as free and subsequently cleared. Between logons, the handle table shared memory and Desktop shared memory for the WinLogon and Applications Desktop is not deallocated. Deallocation is only done at reboot. The ScreenSaver Desktop shared memory (if it exists) is deallocated, and the Desktop data structure(s) are destroyed when the user logs off.

Figure 4.20 depicts the WindowStation and Desktop data structures. Win32 objects are discussed further in the next section.

4.2.6.2 Win32 Objects

Win32 exports two objects: WindowStation and Desktop. Win32 objects:

- Are created by WinLogon in conjunction with Win32 during initialization,

- Are part of the address space of Win32,
- Are referred to by a handle, and
- Have an associated security descriptor.

Win32 objects and handles are not the same as executive objects and handles.[33] When a Win32 object is created, an entry that describes the object, called a *handle*, is added to the handle table. The handle table entry includes a pointer to the security descriptor and the PID of all Win32 client processes that can use that handle. The security descriptor for WindowStation and Desktop objects has the same format as the data structure used for executive objects. The only difference is that the Win32 objects have a logon SID that identifies the logon session rather than a group SID, as previously described. It is the security descriptor that allows the client to get a handle. Once the client has the handle, the granted access mask in the handle is used for subsequent access control decisions.

Win32 also exports a number of sub-objects, called *object attributes*, which are also referred to by a handle, and thus have an entry in the handle table. However, object attributes do not have their own security descriptor but are secured by the security descriptor on the object to which they are associated. WindowStation attributes are *clipboards* and *global atoms*. Desktop attributes are *Windows*, *Menus*, *Hooks*, *DDE Conversations*, *Cursors*, *Icons*. Win32 objects and object attributes are depicted in Figure 4.20.

WindowStation objects can only be created by protected servers (e.g., Winlogon as described above) or by Administrators. A WindowStation has two attributes, a *clipboard* and a set of *global atoms*. A clipboard is a mechanism for sharing any kind of data between threads of the same client and threads running with the SYSTEM token. A global atom is a string which can be shared by different clients. The security descriptor on the WindowStation provides access to these attributes. For the interactive WindowStation object, both of these attributes are cleared at logoff.

A WindowStation is a container for Desktop objects. The security descriptor on the WindowStation indicates that only the WinLogon process can create a Desktop object, as previously described. Having access to the WindowStation does not guarantee access to the Desktop objects which the WindowStation contains. A Desktop has its own security descriptor. Desktop attributes are described in the following paragraphs.

4.2.6.2.1 Windows and Menus Windows and Menus are allocated out of Desktop shared memory. Access to Windows and Menus is controlled by the security descriptor on the Desktop. The handle for a Window or Menu points to the handle for the Desktop. When the Window or Menu is opened, an access check is performed via a call to the Security Reference Monitor (SRM). If the handle for the Desktop is removed from the handle table, the handles for all attributes of the Desktop are also removed from the handle table, and any storage is deallocated.

4.2.6.2.2 Hooks, Cursors, and Icons Hooks are allocated out of Desktop shared memory. A Hook is a way for a thread to tell Win32 which routine to invoke when a certain event occurs. There are a number of different Hook types which can be used to define an event. A thread can register a Hook with Win32 if the thread has access to the Desktop. A Hook can be set either for the entire Desktop or for a single thread of the client process. Win32 stores the Hook information in server memory in two lists as part of the object definition for the Desktop—one for Hooks that apply to all events in the Desktop, and one for thread-specific

[33] Executive objects are created by the Object manager and executive object handles are stored in kernel address space in the handle table of the process that created the object (see Section "Object Manager," on page 58 for additional discussion on executive objects).

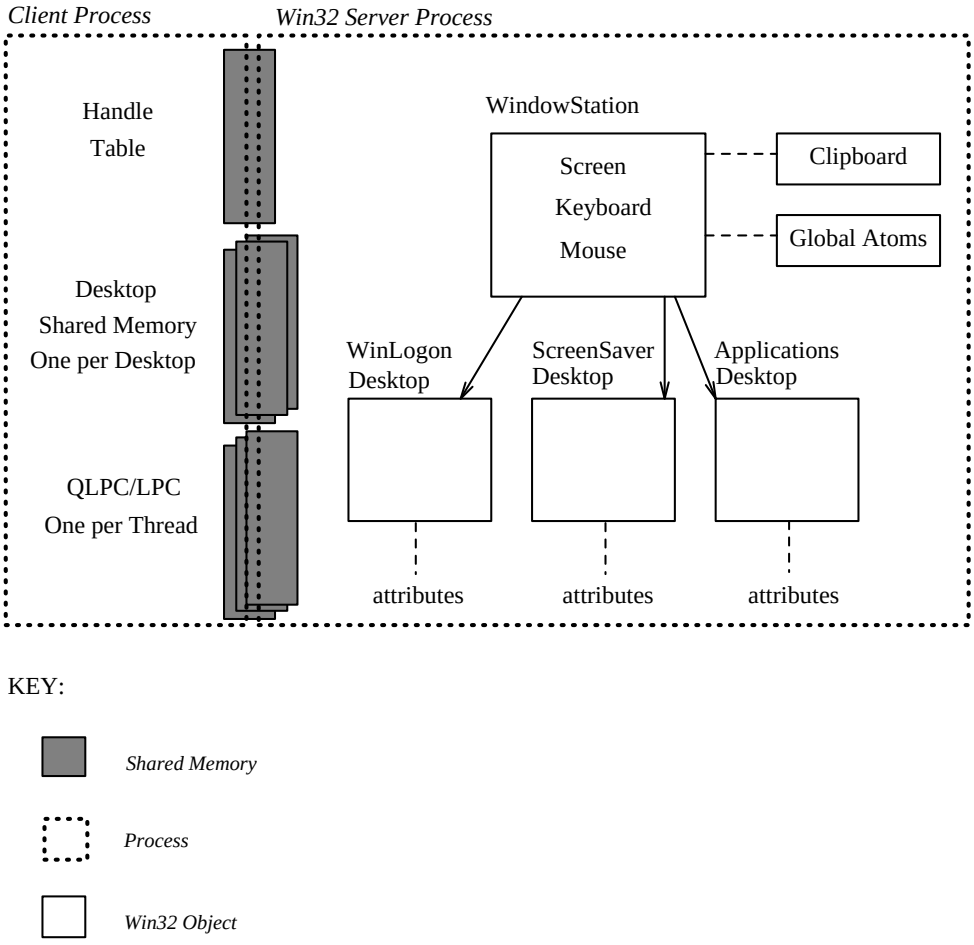


Figure 4.20. Win32 Objects and Their Attributes

events. A given event can have multiple hooks. Processing of hooks can end at any time or can proceed to the end of the list.

Cursors and Icons are very similar, both define images on the Desktop which are stored in Win32 memory. The difference between an Icon and a Cursor is that a Cursor has a "hot spot" which defines the pixel location of the Cursor on the Desktop.

4.2.6.2.3 DDE Conversations Dynamic Data Exchange (DDE) is a protocol for allowing Win32 clients to share data and execute commands. A client process, called the DDE client, can send an initiate message if the DDE client has the DESKTOP_CREATEWINDOW access for the Desktop. The initiate message includes the name(s) of other Win32 client(s), called the DDE server(s), to which the DDE client wants to start up a DDE conversation. The message is sent to the Win32 server who broadcasts the message to all Win32 clients. Any Win32 client identified in the initiate message can respond by sending an acknowledge (ACK) message back to the DDE client via the Win32 server. The Win32 server forwards all ACK messages

to the DDE client, captures the DDE client's token and saves it for subsequent impersonation. At this point a DDE connection has been established and a conversation can begin.

The DDE client can then send messages to a DDE server via the Win32 server. Win32 simply passes the message on to the appropriate DDE server. Subsequently, the DDE server sends an "impersonate client" message to the Win32 server who creates a thread in the DDE server that impersonates the DDE client. This thread can then satisfy the DDE client's request.

DDE conversations are an impersonation mechanism supported by the Win32 server, and an alternative method to the Executive's LPC facility for transferring an impersonation token. A DDE client requests an impersonation level via its Security Quality of Service (SQOS) parameter as with LPC (see Section "LPC Impersonation," on page 86). As with LPC, the DDE server can, as a result of the requested impersonation level, receive an impersonation token for the client. The difference between DDE and LPC impersonation is that the Win32 server, rather than the executive, passes the token to the server side of the connection. Since Win32 has access to all of its clients' processes, it is able to pass the impersonation token by copying the client token and inserting it (with the appropriate impersonation level set) into the server process' thread (see Section "Impersonation," on page 70).

4.2.6.3 Win32 Interface

4.2.6.3.1 Communication Mechanisms Win32 clients communicate with the Win32 server by one of two methods: the Local Procedure Call (LPC) facility or the QLPC facility (see Figure 4.21). The LPC facility is an Windows NT executive mechanism that provides a means for a client and server to communicate. Although LPC allows three methods for exchanging messages, the Win32 server and its clients exchange messages via a shared section (a block of shared memory) which is mapped to the address space of both the client and server (see Figure 4.20). Each thread of the client communicates with the Win32 server via its own section, i.e., each thread/server connection communicates via a private section (see Section "Passing a Message in Shared Memory," on page 84).

QLPC is a message passing facility that is used exclusively by Win32 allowing its clients to achieve greater performance, especially for graphics operations. QLPC is not an Windows NT executive mechanism, although executive objects are used to implement it. To use QLPC, a Win32 client thread sends a request to establish a connection to the Win32 server's connection port. Once the connection is established, the connection port is bypassed. Win32 does the following when it receives a request to establish a QLPC connection:

- Creates a dedicated thread to handle subsequent requests on this connection,
- Allocates a 64 KB section object for passing messages,
- Creates an event pair.

The client thread and the associated Win32 dedicated thread keep a pointer to the shared section and the handle to the event pair in their local memory. The client thread waits on one half of the event pair while the Win32 server is processing a request and the Win32 server thread waits on the other half of the event pair while waiting for the client. The shared section has a header used by both the client and server to determine where to store and find messages. The QLPC message has a fixed structure known to both client and server. This message describes which Win32 server function is being requested, data or other parameters, return results, and other information about the message itself such as the length.

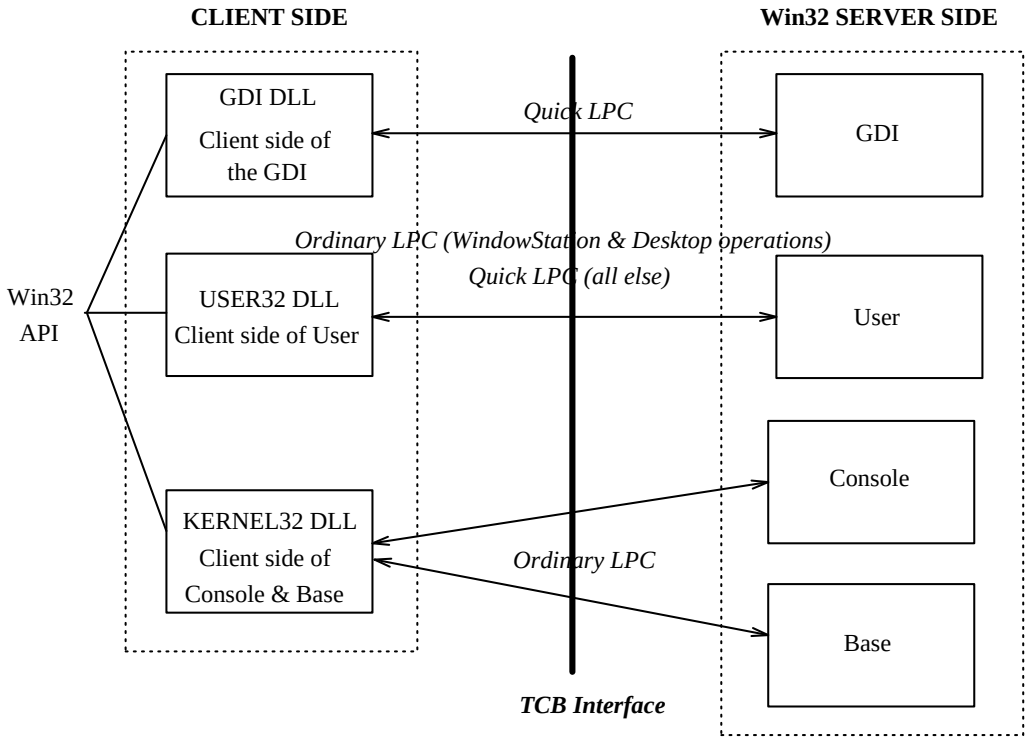


Figure 4.21. Win32 Server and Client Communication

Once the connection has been established, the client sets up the QLPC message in the shared section, signals the Win32 server using the event pair, and waits on the other half of the event pair for a reply. The Win32 server is awakened when its half of the event pair is signaled, processes the request found in the shared section, puts any results in the QLPC message, and signals the client.

As is true for LPC, each thread of the client must establish its own QLPC connection.

4.2.6.3.2 Win32 Server Side and Client Side Architecture Architecturally, the Win32 server is divided into four functional areas: *GDI*, *User*, *Console*, and *Base*. To access the services provided by Win32, clients link to a set of three Dynamic Link Libraries (DLL): *GDI DLL*, *User32 DLL*, and *Kernel32 DLL*. Collectively, the API provided by these libraries are called the *Win32 API* (see Figure 4.21). Only the User32 DLL provide security relevant API.

The API provided by the DLL are either:

- Completely implemented in the DLL code itself
- Wrappers for calls to the Windows NT executive system services or wrappers for calls to other protected servers
- Calls to the Win32 server via an LPC (User32 & Kernel32) or QLPC (User32 & GDI) connection
- State dependent - sometimes they execute within the library code and sometimes they call back to the server.

The TCB interface to the Win32 server is not the DLL API. Instead, the Win32 server TCB interface consists of the parameters passed between the client DLLs and Win32 server process via the communication mechanism, either LPC or QLPC. The information passed between the client and server is not advertised or documented. Instead, the DLL API identify the documented interface to the Win32 server. The following paragraphs describe the four functional areas of the Win32 server. For each functional area, the security-relevant Win32 server interfaces are described from the perspective of the associated DLL API. In addition, the role of the server in providing each security-relevant function is discussed.

4.2.6.3.2.1 Graphics Display Interface (GDI) The Graphics Display Interface (GDI) of the Win32 server and the GDI library code provide support for graphics displays. The Win32 server exports a set of display attributes,[34] e.g., palettes, fonts, brushes, through its API. Clients call the GDI DLL to create and use these attributes, which are the building blocks for all graphics displays. The Win32 server is responsible for making the real I/O video calls to the hardware display.

When Win32 creates a handle for a GDI attribute, it records the handle in the handle table along with the PID of the client that issued the create. The server returns this handle to the client. The client stores the handle in a table in the client's local address space, and records the table index. Subsequently, when the client wants to reference a display attribute, it uses this table index to get the handle and send it to Win32. The Win32 server looks up the handle in its handle table and compares the PID of the client that sent the handle to the PID of the process that created the handle, which is stored in the handle table. The Win32 server rejects the client's request if the two PIDs do not match.

The data associated with display attributes are not shared among clients. Each client has its own handle table in its own address space, which is different than the handle table in Win32 server address space. The server handle can only be used by the client that created the handle. Since display attributes and their associated handles are not shared objects, the Win32 server GDI interface is considered not security relevant and is not described further in this report.

4.2.6.3.2.2 User Interface The User portion of the Win32 server and the User32 DLL provide support for WindowStations, Desktops, Windows, Menus, Hooks, Cursors, Icons, and DDE conversations. As previously described, WindowStations and Desktops are the only objects exported by Win32. The operations that manipulate WindowStations and Desktops are considered security relevant and listed in Table 4.13.

4.2.6.3.2.3 Console and Base Interface The Kernel32 DLL provides the Console APIs and the Base APIs. The Base portion of the Win32 server exports Windows NT executive objects such as events, files, and so on. The Kernel32 DLL implements the Base APIs which provides clients with access to executive objects exported by Win32.

The Console APIs and the Console functions provided by the Win32 server, support the DOS command mode user interface. These API provide no additional access to system resources and are simply provided for emulation of 16-bit DOS applications.

[34] These are GDI attributes and should not be confused with WindowStation and Desktop attributes.

Function
Create a new Windowstation.
Disable the ability to switch and/or open a Desktop while critical security operations are underway.
Unlock the WindowStation and enable the capability to switch and open a Desktop.
Check the access permission on the WindowStation and return a handle if the access check passes.
Close a reference to a WindowStation.
Assign a WindowStation to a process.
Register with Win32 as the Win32 logon process.
Associate the calling process with a Logon window.
Create a Desktop.
Return a handle to the Desktop currently receiving input.
Check the access permission on the Desktop and return a handle if the access check passes.
Close a reference to a Desktop.
Open the WindowStation and Desktop when a client application makes the first User32 or GDI call.
Assign a thread to another Desktop by handle.
Log out and shut the system down.
Set the level of impersonation.
Store a pointer to the DDE Client in the DDE Server's context.
Set up DDE Server to use DDE Client's token.
Get the security attributes of a Desktop or a WindowStation.
Set the security attributes of a Desktop or a WindowStation.

Table 4.13. Security Relevant User 32 DLL Functionality

4.2.7 Spooler

The Spooler is a protected server whose major functions are the spooling and printing of print jobs. The Spooler runs in the SYSTEM context, with the SYSTEM token (see Table 4.9, on page 107). The Spooler provides functions to manipulate printers and jobs. To support its functionality, the Spooler exports three objects: server, printer, and job. These objects are hierarchical in that the server object contains printer objects and printer objects contains job objects. The server's DACL contains ACEs marked for InheritOnly, ContainerInherit, and ObjectInherit (see Section "ACL Inheritance," on page 146 for a description of container objects and these inheritance flags). Therefore, the printer object inherits ACEs from the server object and the job object inherits ACEs from the printer object. The access rights available for each of these objects is described in Section "Overview of Object Access Rights," on page 142. Each object is described below.

The Spooler calls the SRM for access to and auditing of its objects. If access validation succeeds, the granted access mask is stored in a handle to the object. Subsequent accesses to that object are checked against the handle. This handle is not the same as the executive handle, but is similar in structure and use. For each Spooler object opened, this handle associates the process that requested the open with the object and contains the granted access.

For each port (e.g., LPT1), the spooler has created and connected a physical printer to, the Spooler has a thread of execution called a printer thread. The Spooler will also have one scheduler thread responsible for

coordinating and synchronizing the printer threads. The scheduler thread is activated whenever the state of an existing print job changes (a new job is spooled, the state of the printer object changes, or a new printer object has been added). The scheduler thread is also activated when a printer thread completes its print job and is ready for another print job. When a printer object is removed, the scheduler is responsible for signaling that printer thread to be terminated. When a print job has completed, the scheduler must decide what job the freed thread can print next. If no job is available, the printing thread is instructed to wait until one is. To determine what print jobs a printing thread can print, the scheduler examines all the printers connected to the port that the printer thread is responsible for, as well as all the jobs for those printers. The job will be chosen based on the specific criteria such as printer paused or not, printer and job scheduling times allow printing, etc.

The spooler writes the security descriptor information for each printer in the registry when the printer is created and when the printer's security descriptor is altered.

When the Spooler is brought up, it reads the registry and re-associates the security descriptor information with each printer.

4.2.7.1 Server

There is only one instance of the server object. Access to the server object allows for: the installation and de-installation of a printer driver (e.g., postscript driver); the creation, deletion, enumeration, and configuring of ports (where the physical printer is connected, e.g., LPT1); and the creation, deletion, and enumeration of printer objects.

Creating and opening a printer object requires ServerAccessAdminister access to the server.

4.2.7.2 Printer

The printer object is a container object which inherits access rights and flags from the server object's security descriptor. It contains job objects. A printer object must exist for each physical printer to which jobs will be submitted. There can be many printer objects for one physical printer. The printer object can be considered a queue for jobs to be printed. As stated above, ServerAccessAdminister access to the server is required to create printer objects. However, the Spooler provides functions to manipulate the printer object based on the access rights to the printer object. Printer manipulation functions provided include: pausing, resuming, deleting, connecting to, disconnecting to, setting, and enumerating. The functions provided which require PrinterAccessAdminister are to pause, purge, resume, and change the settings of a printer. Reordering a job on a specified printer object also requires PrinterAccessAdminister access to the printer object. Adding a job to specified printer object (submitting a job for printing) requires PrinterAccessUse to the printer object.

4.2.7.3 Job

The job object is contained within the printer object and inherits ACEs and flags from the printer object's security descriptor. Each job object corresponds to a job submitted to the spooler for printing. The spooler provides functions to manipulate jobs. Deleting a job on a specified printer requires delete access to that job. Pausing and resuming a job on a specified printer requires JobAccessAdminister to that job. After adding a job to a specified printer, the Spooler returns a JobID which is a unique number for that printer

identifying that job. The JobID is used in service request to the Spooler to manipulate the job. A user will have JobAccessAdminister access to only their jobs.

4.3 Administrator Tools

The Administrator tools run in user mode. However, unlike servers, these tools execute in the security context of the user. Some tools require the user to possess one or more privileges (which is enforced by the underlying server or executive subsystem, and not the admin tool itself). Since privileges are associated with users and their tokens, and not with executable programs, the Administrator tools are only in the TCB because a privileged user must use them to manage the TCB.

The set of administrative tools included in the evaluated configuration are Backup, Chkdsk, Control Panel, Disk Administrator, Event Viewer, File Manager, Print Manager, Program Manager, Registry Editor, Setup, and User Manager.

4.3.1 Backup

Backup is a tool used to protect data from accidental loss or hardware and media failures. It provides functions to back up disk files to tape, restore tape files to disk, label, track and identify tapes. Members in the Administrators or Backup Operators groups have Backup and Restore privileges allowing them to bypass the protection provided by DACLs.

File permission information is stored on tape with backup files by default. The backup operator has the option to restore the original file access permissions. If this option is not selected, the files inherit the permission of the directory into which they are restored.

4.3.2 Chkdsk

Chkdsk is a command that checks for and displays file system integrity problems. It is security relevant in Windows NT because it also examines the security descriptor information of NTFS partitions. This is also done automatically during system startup by a process called autocheck.

4.3.3 Control Panel

Users can customize and administer Windows NT with the Control Panel. The following list includes those Control Panel functions that are security relevant. Their use requires privileges.

- Date/Time: Changes the date/time and the time zone of the computer clock.
- Devices: Starts, stops, and configures the startup type for device drivers.
- Drivers: Installs, removes, and configures drivers.
- Ports: Changes communication settings on serial ports.
- Printers: Opens Print Manager.

- Services: Manages services defined in the Registry (i.e., start or stop the services available in the computer and configure the startup options for each service).
- System: Specifies the startup Operating System, sets system and user environment variables, changes system performance parameters, determines error recovery procedure, and defines paging file size.

4.3.4 Disk Administrator

The Disk Administrator provides functions to partition disks, create and delete volume sets, extend volume sets, create and delete stripe sets, establish and break mirror sets, and recover data.

Volume sets and stripe sets are mechanisms whereby free space from multiple disk partitions are combined into a new partition. A user must be a member of the Administrators group to start this tool.

4.3.5 Event Viewer

The Event Viewer is used to view and manage event logs, including the security log. It allows for viewing, sorting, filtering, and searching the event logs. The user must have access to the event log file in order to successfully view it. To view the contents of the security log, the user must be logged on as a member of the Administrators group. Use of the event viewer itself requires no privilege. Security is enforced by the DACL on the log.

Event Viewer provides two sorting options: newest events first or the oldest events first. To filter events, there is a predefined set of options available. Some of the filter options are: from date, through date, warnings, errors, success or failure, source of logging events, user, and event category (e.g., policy changes). Event Viewer also provides for the export of audit data in a number of formats, including comma-delimited ASCII.

4.3.6 File Manager

The File Manager provides file and directory manipulation and organization functions. It provides functions such as: create and remove directories; move, copy, and delete files; secure files and directories; and perform other disk, directory, and file management tasks. Some functions are restricted to certain groups. Manipulation of directory and file objects is controlled by their DACLs, which are set by using File Manager.[35]

The file manager provides functions to audit operations on files and directories. To enable auditing of files or directories, membership in the Administrators group or possession of the security privilege is necessary. Using the File Manager, an administrator can enable different types of auditable events on the files and directories when these events are generated by specified users and groups. To change permission on a file not owned by the user, the TakeOwnership privilege is required.

4.3.7 Print Manager

The Print Manager provides functions to manage printing, printers, and print jobs. The major functions it provides are: installing printers, changing a printer's properties, and setting event types for auditing.

[35] Only directories and files on drives formatted for NTFS can be protected through the DACL mechanism.

Some functions, such as creating a printer or changing the printer properties, are restricted to members of the Administrators or Print Operators group. One can also change printer properties if one is granted Full Control access to the printer. Printing properties include: hours available to print, printer priority, the separator page option, and the spooling option. A printer is secured by a DACL, and auditing of printer access is controlled by a SACL. The print manager provides the interface whereby SACLs and DACLs are modified. To change the DACL or SACL on a printer object, one must be the owner of the printer or have been granted Full Control access, or have the TakeOwnership privilege.

A user can always control printing of his own print jobs. However, members of the Administrative and Print Operators group can control the printing of other users' print jobs. Manage Documents or Full Control access to the printer also allows the control of other users' print jobs.

4.3.8 Program Manager

The Program Manager is the basic user interface that is used to organize the applications and files in groups and to start the applications easily. The Program Manager can also be used to initiate logging off or shutting down the computer. The Program Manager provides several methods for launching applications and switching between them. To start an application one may either click on its icon in the program manager, or enter its command line in the "Run" dialog box. The Task List provides similar capability and also allows users to see a list of ready tasks from which to choose.

4.3.9 Registry Editor

The Registry Editor is a tool that can be used to view and edit the configuration database of a Windows NT system. It does not appear in any default program groups in Program Manager, although it is installed automatically when Windows NT is installed. Most often, changes to the configuration database are accomplished through one of the other tools but certain changes can only be accomplished through the Registry Editor. For example, the uses for the Registry Editor mentioned in the TFM:

- Disable the OS/2 and POSIX subsystems
- Force Windows NT to halt when it cannot generate an audit event record
- Allow the audit of backup and restore rights (See Section "Privilege Use," on page 162)
- Require users to log on before shutting down the computer
- Prevent display of a username in the logon dialog box
- Display a message box before user logon

The Registry Editor is invoked by executing the file REGEDT32.EXE. Keys modified by the Registry Editor are protected by DACLs; to change a key, requires the TakeOwnership privilege or ownership of the key. To change auditing of keys requires the Security Privilege.

4.3.10 Windows NT Setup

Setup is the same program used to accomplish the original installation of Windows NT and allow administrators to change the configuration in a number of areas. Specifically, Setup permits the installation of

drivers for SCSI devices, keyboards, mice and tape devices. In addition, Setup allow administrators to revise their choices of "Windows components," the suite of ancillary software that comes with Windows NT, such as screen savers, calculator and background designs.

4.3.11 User Manager

The User Manager is used to create and manage user accounts and groups, and implement the security policy. The actions a user can perform in User Manager is determined by the group memberships of his user account. If the user does not have sufficient authority to perform an action in User Manager, then that command is shown to be unavailable. For a list of Built-in groups and their descriptions refer to the Table 5.1. In particular, a user who is a member of the Administrators group can perform all functions of the User Manager. A user who is a member of the Power Users group can create user accounts and groups, and can modify and delete those user accounts and groups, but he cannot modify or delete accounts or groups that he did not create. A Power User can also add users to the Power Users, Users and Guest groups. A user who is a member of the Users group can create groups, modify and delete these groups, and give any user membership in these groups.

The functions the User Manager provides for implementing the security policy are: controlling the way passwords may be used by all user accounts, controlling the privileges assigned to groups and user accounts, and controlling the audit policy by defining the security events that will be logged. These functions are restricted to members of the Administrators group.

4.3.12 User Profile Editor

The User Profile Editor is an administrative tool for creating a customized user profile. A customized user profile differs from the user profiles that Windows NT automatically creates and maintains in these ways:

- A single user profile can be made available to a user from any computer on the network.
- A single user profile can be shared by a group of users who perform similar tasks.
- A custom user profile can limit a user's ability to change the user profile and to gain access to some commands in Program Manager.

A user profile is a definition of the configuration for a specific user or group of users. By default, the system maintains a profile for each user who has logged on to the computer, except for Guest accounts. Each user profile contains information about particular user's Windows NT configuration. Much of this information is about things the user can set, such as color scheme, screen savers, and mouse and keyboard layout. Other information is about things that can be set only by an administrator, such as access to common program groups or network printers.

4.4 Initialization and Shutdown

Windows NT startup has four main parts, as listed below. A new thread is created for each part (executed sequentially rather than in parallel).

- The Boot Loader Program.
- Phase 0 Initialization: Start the Microkernel and some parts of the Executive.
- Phase 1 Initialization: Complete the initialization of the Executive.
- Phase 2 Initialization: Start the Protected Servers.

4.4.1 Boot Loader

The Boot Loader loads the first layer of software needed to start the operating system. It receives from firmware the names of the console input device, the console output device, the load file name, the system device name, the system directory path for booting, and the Registry System key. Using the load file name and the system device name, the boot loader constructs the path name to the system image and loads it into memory. It also starts *NtLdr*, which displays a menu offering possibly several installed versions of Windows NT for booting.

The boot loader then reads the system configuration information stored in the Registry tree directory (see Section "Configuration Manager," on page 102) and constructs the path name to the HAL code and to a mini file system driver, and loads them into memory. It also loads the HAL.DLL and any other DLLs which it or *ntoskrn* reference. The Boot Loader then loads and initializes all the device drivers and other programs listed in the Registry for automatic startup at boot time.

4.4.2 Phase 0: Initialization

The boot loader, running in the context of the boot process, calls a microkernel routine to set up and initialize architecture-dependent data structures. The microkernel then initializes its architecture-independent data structures, including the system time structure and the thread table and its associated structures, and then initializes the idle thread process object and the idle thread object. This idle thread is scheduled for execution at the highest priority level. Then the microkernel calls a routine to begin the initialization of the rest of Windows NT Executive.

During the Phase 0 Executive Initialization process, the Windows NT Executive Subsystem components initialize their data structures, which are needed by other components of the Executive Subsystem. For example, the Object Manager creates and initializes data structures that allow Windows NT Executive Subsystem components to create objects. The following sequence of events take place during Phase 0 initialization.

- The Memory Manager initializes its synchronization objects; builds paging support structures such as page tables, PFN databases, and paged and non-paged memory pools; computes the number of pages needed for the system image; maps the code, data, stack, and trap handlers for the executive; and performs architecture-dependent initialization of memory relevant data structures.
- The Executive Heap Manager performs its Phase 0 initialization.
- The Object Manager initializes the synchronization objects it needs to protect its data structures and allocates a non-paged memory pool to store the non-paged object headers for the objects that are in the paged pool. It also creates the object types it exports.
- The Security Reference Monitor begins its Phase 0 initialization.

It initializes the LUID allocation services; allocates and initializes the global security variables, Universal (Well-Known) SIDs and Windows NT-defined SIDs (see Section "Identification and Authentication," on page 181), System Default DACL, and known privilege values; initializes the Reference Monitor data structures; and creates a Token Object Type object and a System Token, and assigns this token to the boot thread object.

- The Process Manager sets the minimum and maximum Working Set size and the default page file limit; creates the Thread Object Type and Process Object Type objects; initializes the active process list; and creates a System Process, assigns the System Token in the boot thread to it, and creates a Phase 1 initialization thread in it.

The microkernel then lowers the priority of the idle thread, causing the newly created Phase 1 initialization thread to be scheduled for execution.

4.4.3 Phase 1: Software Initialization

Phase 1 initialization builds on the structures that are in place after the completion of Phase 0 initialization. The components of the Windows NT Executive will be fully functional at the end of Phase 1 initialization.

Phase 1 software initialization begins when the Phase 1 initialization thread begins execution. This thread first boosts its priority so that it will not be pre-empted.

The Phase 1 initialization thread calls the HAL to query the real-time clock and set the system time. In a multi-processor environment, this thread starts other processors, using HAL in an architecture-dependent manner, and establishes affinity threads so that these processors can be used. The Object Manager is started.

The Object Manager performs its Phase 1 initialization. It creates the root directory, \, and the directory for Executive object types. The object Name Space (see Section "Object Manager," on page 58) is opened and entries are made into this directory for all the objects that have been created so far. The Phase 1 initialization thread creates several worker threads to service Executive work queues. Several other object types such as Event Object Type are instantiated. The root directory objects`\ArcName`(for Alpha Processors only), `\Device` and the symbolic link to `\SystemRoot` are created.

The Memory Manager performs its Phase 1 initialization. It removes the executive components that can be paged, the boot driver list, HAL code, and system hive buffers from memory; counts the number of system-wide pages available to be locked for I/O; creates an event object to be signaled when a paging file is created; and creates a modified page writer thread, two balance set manager threads and a synchronization event object for these two threads. The initialization thread calls the Cache Manager and the I/O Manager to initialize their respective data structures. The Configuration Manager is called to perform Phase 1 initialization. It creates a worker thread synchronization object, a Key Object Type object, and the SYSTEM and the HARDWARE hives, which are filled with data gathered by the loader during Phase 0 initialization.

The I/O Manager, the LPC Facility, the Process Manager, and the Security Reference Monitor are called to perform Phase 1 initialization. The Process Manager maps the NTDLL.DLL to user address space for use by user-mode processes. The Security Reference Monitor creates an LPC port, an event to synchronize with the LSA, and a server thread to service commands from the LSA. The Executive components that are needed to start the Session Manager have now been started and initialized. The Session Manager is started, using information contained in the SYSTEM hive.

Finally, the Phase 1 initialization thread deletes the initialization code and calls the Memory Manager to change itself into the "zero page" thread.

4.4.4 Phase 2: Software Initialization

After the Windows NT Executive is initialized and the Session Manager has started, initialization of essential user-mode processes is begun.

The Session Manager creates a Session Manager LPC port and two service threads to service that port; reads the entries in the registry to load Win32; and starts WinLogon, which in turn registers itself with Win32 as the logon process.

WinLogon establishes an LPC connection to Win32 and starts two other protected servers: the Service Controller and the Local Security Authority(LSA)/Security Accounts Manager(SAM) process. It then establishes an LPC connection to LSA/SAM server which in turn establishes an LPC connection to the Security Reference Monitor. Winlogon starts these two servers because it requires authentication through the LSA/SAM server and because the Service Controller immediately starts the Event Logger, which will receive audit event records from the LSA/SAM server.

The Service Controller now reads the Registry entries and, depending upon the instructions contained in the Registry and the user API calls, will start and control the listed services. The Event Logger service is then started.

When all services are started, Windows NT displays the "Welcome" screen. A user can now logon to the system.

4.4.5 Shutdown

Shutdown occurs in two phases: user-mode process shutdown, then executive shutdown.

4.4.5.1 User-Mode Shutdown

Users can shutdown the system from the File menu in Program Manager or by pressing the SAS and clicking on Shutdown. The "Shutdown Computer" dialog box is displayed. The user has the option of shutting down the computer or shutting down the computer and restarting. Normally, shutting down the computer without specifying restart would precede turning off the computer. WinLogon calls ExitWindowsEx with the EWX_SHUTDOWN or EWX_REBOOT argument. ExitWindowsEx notifies the protected servers and services (LSA/SAM, spooler, etc.) that shutdown is occurring. When control returns to WinLogon it performs the final system shutdown by using executive services.

4.4.5.2 Executive Shutdown

The executive provides services to support system shutdown in one of two ways:

- Shut the system down and automatically reboot.

- Shut the system down and halt for power-off.

The shutdown procedure determines that one of the above modes was specified and that the caller has Shutdown privilege. If so, it informs the I/O Manager that a shutdown is in progress. The I/O Manager calls the shutdown handlers of all of the registered drivers that have requested shutdown notification. The shutdown procedure calls the Configuration Manager to flush the hives to disk and the Memory Manager to flush all dirty pages.

The shutdown procedure invokes the I/O Manager again, which notifies all of the active and registered file systems that the system is going down. Each file system flushes its caches and then passes the shutdown IRP to the drivers below it, and each driver flushes its caches and passes the IRP to any lower drivers. This continues until the IRP is processed by all of the drivers under all of the file systems.

The function then either returns (no reboot), or it calls the HAL at the appropriate restart routine.

This page intentionally left blank

Chapter 5

Security Architecture

This chapter describes the security architecture of Windows NT. This chapter assumes the reader is familiar with the information provided in Section "Software Architecture," on page 49.

There are two principal sections in this chapter. The first section describes all the protected resources in Windows NT; specifically users and user accounts, subjects, and objects. The second section describes the Windows NT security policies; specifically discretionary access control, identification and authentication, privileges, object reuse, and auditing.

5.1 TCB Protected Resources

This section describes user and group accounts, subjects, and objects for Windows NT, along with the security-related attributes of each.

5.1.1 Users and Groups

In Windows NT users, and groups of users, are represented by *accounts*. Windows NT supports three types of accounts:

- User Account: Represents a single, human user. Information included within a user account includes user name, associated groups, password, and home directory.
- Global Group Account: Represents a list of user accounts. This type of account is only available in the Windows NT Server product.
- Local Group Account: Represents a list of user or global group accounts.

The distinction between local and global group accounts becomes more obvious in networked configurations.[1] For a stand-alone system, the only difference between the two types of group accounts is that a local group may include both user and global group accounts, whereas a global group can only include user accounts.

Except for information about privileges, all information for each of these accounts is maintained by the Security Account Manager server (SAM, see Section "Security Accounts Manager," on page 112).

The Local Security Authority (LSA) server maintains all information describing privileges assigned to accounts.

[1] Networked Windows NT systems are not within the evaluated configurations.

All three types of accounts have a unique identifier called a Security Identifier (SID). A SID representing any of the three types of account may be used for access control.

5.1.1.1 Built-in Accounts

Windows NT is delivered with built-in accounts for all three types of accounts. There are two built-in user accounts:

- Administrator Account: This is an "all powerful" user account that can never be disabled nor removed. It is the only enabled user account when the system is initially installed.
- Guest Account: This is an ordinary user account with no special group membership.

 In Windows NT Server, the guest account is disabled by default and cannot be used to logon. In Windows NT/WS, the guest account is *enabled* by default *without* a password. The TFM informs the system administrator to disable this account before allowing users to use the system.

There are also two built-in global group accounts (global group accounts are only defined for the Windows NT Server product). Both of these accounts are intended to support networked configurations (although they can also be used in stand-alone configurations).

- Domain Admins Account: Initially includes only the Administrator user account.
- Domain Users Account: Initially includes both of the built-in user accounts (Administrator and Guest).

There are a number of built-in local group accounts, and they differ between the Windows NT Server and Windows NT/WS products. Table 5.1 lists all the built-in local group accounts, and identifies those accounts that have security-related administrative capabilities. By default, all of the built-in local group accounts have no members except for Administrators (which includes the Administrator user account and, in the Windows NT Server product, the Domain Admins global group account), Users (which includes the Domain Users global group account), and Guests (which includes the Guest user account). See Section "Privileges," on page 151 for the list of specific system privileges assigned to these accounts.

In addition to the built-in accounts, there is also a well-known, system-defined SID called *Everyone*. While Everyone is not technically a group, by definition it corresponds to all users and groups, and can be used like a group SID (e.g., in DACLs, as an owner).

5.1.1.2 User Logon and Authentication

In Windows NT, there are three forms of user logon supported: interactive, service, and network. An *interactive* logon is via WinLogon and is the typical form of user logon (see Section "WinLogon," on page 115). A *service* logon is a batch logon facility and is supported via the Service Controller (see Section "Service Controller," on page 116). The third form of logon, *network*, is not supported in the evaluated configuration.

Local Group	Product		Security Relevant	Description
	NT Server	NT/WS		
Administrators	X	X	yes	This is the most powerful group available to users. Members of this group can perform all security-relevant administrative functions.
Backup Operators	X	X	yes	This group is intended for users with responsibility to back and restore file systems. Members of this group received the Backup and Restore privileges (which when enabled, effectively overrides all DACL checks for read and write access).
Server Operators	X		yes	This group, which is only available on the Windows NT Server product, is intended for users who are responsible for all security administration, except for user and group accounts administration, of a Windows NT Server system. Members of this group can perform most administrative functions (except for account manipulation) to include backup and restore.
Account Operators	X		yes	Account operators can manage most aspects of normal user and group accounts, including creating and modifying user and group accounts. Account operators cannot modify user accounts which are members of the Administrators group, nor change any information (including membership) about the security-related built-in accounts.
Print Operators	X		yes	Members of this group can create new printers, and manage existing printers.
Power Users		X	yes	This group is fairly powerful, but only exists on Windows NT/WS. Members of this group can create new user and group accounts (but only change information for those accounts they create), change the system time, and manage and create printers.
Users	X	X	no	This is the group to which most ordinary users are assigned. In Windows NT Server, members of this group are not allowed to logon to the system (this group is meaningful in networked configurations for Windows NT Server). In Windows NT/WS, members of this group may logon.
Guests	X	X	no	This group is intended for users with very limited access to the system. In Windows NT Server, this group is similar to the user group (i.e., no logon allowed). In Windows NT/WS, members of this group have limited capabilities (e.g., they cannot change their local account information).
Replicator	X	X	N/A	This group is intended to support the Replicator service, which provides a means to maintain consistency between multiple copies of the same files or directories. The Replicator service is only meaningful in a networked configuration and is not part of the evaluated configuration. No users are a member of this group by default, and the TFM advises the administrator not to add users to this group.

Table 5.1. Built-in Local Group Accounts

5.1.2 Subjects

A subject in Windows NT is a process with one or more threads.

A process has a private virtual address space (see Section "Virtual Memory Management," on page 63), a private handle table (see Section "Handles and Handle Tables," on page 61), and a *security context*. All threads within a process have equal access to the address space and handle table of the process.

The security context of a process is represented an access token (see Section "Impersonation," on page 70 and Section "Tokens," on page 70). A token contains information used in access control and privilege checks and includes the user's SID, group SIDs, and privileges. In essence, a token is the system's internal representation of a user's authorizations. Every process has exactly one *primary token*. The primary token represents the user on whose behalf the process is operating (and the user's authorizations). The contents of a primary token are determine as part of LSA's authorization process, based on the user and group account information stored in the SAM database, and privilege assignment information maintained within LSA.

Each thread within a process may also have one additional token, called an *impersonation token*. The impersonation token represents another subject that has authorized the thread to utilize a subset of the subject's authorization (see Section "Impersonation," on page 70 and Section "Tokens," on page 70). Regardless of whether an impersonation token is present, the TCB always knows the primary token of a process, and therefore can always associate the actions of a process with the owning user.

5.1.3 Objects

This section describes Windows NT's protected object types (i.e., all of the resources that are protected by the TCB and made available to non-TCB subjects). The security related attributes for each object type are also described.

5.1.3.1 Object Types

Windows NT object types fall into two broad categories: executive objects and server objects. *Executive objects* are created, managed, and protected by executive (i.e., kernel-mode) subsystems. Security for executive objects is provide via the Object Manager and Security Reference Monitor. *Server objects* are abstractions of TCB protected servers (i.e., trusted user-mode processes). Individual protected servers are responsible for managing and protecting their resources, though protected servers use the services of the Executive's Security Reference Monitor to ensure consistent enforcement of security policy. Table 5.2 lists all Windows NT object types, the Executive subsystem or protected server that manages the object type, and the section of this report that describes the managing subsystem or server.

5.1.3.2 Object Security Attributes

Each Windows NT object potentially has the following associated security attributes:

- *Owner ID:* A SID representing the *owner* of an object. The owner SID may represent a user, a global group, or a local group. The object owner has some intrinsic access to an object. Specifically, the object owner always has *WriteDac* and *ReadControl* (see below) to owned object.

Executive Object Types		
Object Type	**Managing Subsystem**	**Subsystem Report Section**
Event	Executive Object Services	4.1.9, on page 104
Event Pair	Executive Object Services	4.1.9, on page 104
I/O Completion Port	Executive Object Services	4.1.9, on page 104
Key	Configuration Manager	4.1.8, on page 102
Mutext	Executive Object Services	4.1.9, on page 104
Object Directory	Object Manager	4.1.2, on page 58
Object Symbolic Links	Object Manager	4.1.2, on page 58
Port	Local Procedure Call Facility	4.1.6, on page 82
Process	Process Manager	4.1.5, on page 81
Profile	Executive Object Services	4.1.9, on page 104
Section	Memory Manager	4.1.3, on page 63
Semaphore	Executive Object Services	4.1.9, on page 104
Thread	Process Manager	4.1.5, on page 81
Timer	Executive Object Services	4.1.9, on page 104
Tokens	Security Reference Monitor	4.1.4, on page 69
I/O Subsystem Objects (exported as "file" objects)		
Device	I/O Manager and Device Drivers	4.1.7.1, on page 90
Mailslot	Mailslot File System	4.1.7.3.6, on page 101
Named Pipe	Named Pipe File System	4.1.7.3.5, on page 101
NTFS File	Windows NT File System	4.1.7.3.2, on page 96
NTFS Directory	Windows NT File System	4.1.7.3.2, on page 96

User Accessible Protected Server Object Types		
Object Type	**Managing Server**	**Server Report Section**
Log	Event Logger	4.2.5, on page 117
Service	Service Controller	4.2.4, on page 116
Desktop	Win32 Server	4.2.6, on page 118
Print Job	Print Spooler	4.2.7, on page 126
Printer	Print Spooler	4.2.7, on page 126
Print Server	Print Spooler	4.2.7, on page 126
WindowStation	Win32 Server	4.2.6, on page 118

Table 5.2. Windows NT Protected Objects

- *DACL:* A DACL is the principal means by which users control access to objects. DACL entries represent access granted (or explicitly denied) to a user or group of users. Not all object types must have an associated DACL (though all objects allow a DACL). In the case of no DACL, all users are implicitly granted all access to the object (assuming they are able to name or open the object). See below for more about which objects require a DACL.
- *SACL:* SACLs are system administrative structures associated with individual objects. SACLs control object security auditing for the associated object. In order to access an object's SACL, the caller must have a privilege (see Section "Privilege Enforcement and Auditing," on page 152).

A fourth attribute, *group ID*, is also associated with each object. This attribute has no special security meaning and is only provided for compatibility with some operating system standards (e.g., POSIX).

5.1.3.3 Discretionary Access Control List Defaults

All named executive objects must have a DACL. Named executive objects are all those objects with an entry in the global object name space or one of the secondary name spaces supported by Windows NT File System and the Registry (see Section "Global Name Space and Related Objects," on page 60). If, when a named object is created without a supplied DACL, Windows NT will assign an appropriate DACL to the object. This DACL can be inherited from a parent directory, provided by a TCB protected server, or taken from the default DACL in the thread's token. See Section "Discretionary Access Control Lists," on page 146 for more information about DACL assignment.

Except for process, thread, and token object types, unnamed executive objects (i.e., those that do not have an entry in the global or secondary name spaces) need not have a DACL assigned. Unnamed objects are protected through inheritance of object handles (see Section "Handles and Handle Tables," on page 61). Process, thread, and token objects must always have a DACL since they can always be accessed using special APIs (i.e., NtCurrentProcess, NtCurrentThread, NtOpenProcessToken, NtOpenThreadToken).

5.1.3.4 Overview of Object Access Rights

Access to objects is principally controlled via a DACL associated with an object. The types of access permitted (or denied) in a DACL entry, called an ACE, is represented by an *access mask*. The access mask structure is standard for all objects, ensuring that the Security Reference Monitor (SRM) can perform access validation regardless of the specific object type (see Section "Access Mask Data Structure," on page 73 for a description of the access mask data structure).

While the access mask structure is standard, the interpretation of some fields within this structure is type specific. The Executive subsystem or protected server responsible for managing a given object type defines the semantics for the type-specific portion of an access mask, and provides certain mappings to the SRM. The SRM uses this information when performing access validation.

An access mask specifies five types of access rights: standard, specific, generic, maximum allowed (MA), and access system security (AS). MA and AS are only used in a request access mask (i.e., as a parameter to an access request, see Section "Access Mask Data Structure," on page 73). MA is used to request that all allowed access rights to the object be granted. AS is used to request access to the object's SACL (which requires privilege). Standard, specific, and generic access rights, and their relationships, are described below:

1. *Standard Access Rights:* The standard access rights have the same meaning for all objects, regardless of their type. There are five standard access rights:
 - *Delete:* Allows an object to be deleted.
 - *ReadControl:* Allows security-related information associated with an object to be read. This information includes the object owner, the DACL, and group.
 - *WriteDac:* Allows the DACL to be modified.
 - *WriteOwner:* Allows a thread to change the object owner to any of the SIDs(user or group) in the thread's current token. (The TakeOwnership privilege grants this right for all objects regardless of the DACL).
 - *Synchronize:* Allows the object to be used for process synchronization (e.g., to be waited on).

 Except for Synchronize access, all standard access rights have meaning for each object type. Only a few objects (mostly inter-process communication and process objects) support the Synchronize access right.

2. *Specific Access Rights:* An access mask has room for up to sixteen type-specific access rights. These rights have no generic meaning, and are completely type-specific. The semantics for the type-specific access rights are defined by the managing subsystem or server for the object type.

3. *Generic Access Rights:* Generic access rights provide a means to reference specific and standard access rights in a more coarse manner. There are four generic access rights:
 - *GenericRead*
 - *GenericWrite*
 - *GenericExecute*
 - *GenericAll*

 Generic access rights are not actual access rights to an object, but represent a mapping to some subset of the standard and specific access rights. By convention, GenericAll maps to all standard and specific access rights for a given object type.

 The convention for mapping between the other generic access rights and the standard access rights are given below:

GenericRead	ReadControl
GenericWrite	ReadControl
GenericExecute	Synchronize, ReadControl

 The mapping between the specific and generic access rights is defined for each type by the managing subsystem or server. All specific access rights for a given object type need not map to one of the generic access rights (except for GenericAll which always maps to all specific and standard access rights).

5.1.3.5 Object Type-Specific Security Attributes

Object Type:	Name of the object type and a brief description of the object's purpose.
Managing Subsystem or Server:	Executive subsystem or protected server responsible for the object type. See Table 5.2 for references to report section describing the subsystem or server.

Specific Access Rights:	The type-specific access rights defined for the object type, along with a brief definition of the type of access each right allows (or denies).
Mapping to Generic Access Rights:	A mapping between the specific access rights and the generic access rights for this object type. As discussed above, the mapping between the generic and standard access rights is the same for all object types. A mapping to GenericAll is not provided for each object type since GenericAll maps to all specific and standard access rights for each object type.
Default Protection (Servers Only):	Most server object types have pre-defined DACLs and owner for all instances of that type. Often, this DACL cannot be changed.

5.2 TCB Security Policies

This section describes the security policies implemented by Windows NT. These polices are: Discretionary Access Control (DAC), Identification and Authentication, Privileges, Audit, and Object Reuse. These policies address several security concerns.

The DAC policy is concerned with limiting access to information based on the identification of users and allowing a user to specify who has access to information under that user's control. Accountability is the main objective of the Identification and Authentication policy and the Audit policy. Individual accountability is a key concept in securing a system. Identification is made credible by authentication. A unique identification is crucial to providing individual accountability, providing an audit trail which records all security relevant actions of users, and providing support to enforce discretionary access to objects based on identity. Privileges provide exception to some security relevant rule. These exceptions should be controlled and limited to certain users. The ability of a user to obtain residual information from another user is prevented by the Object Reuse Policy.

How Windows NT addresses these concerns is described in this section.

5.2.1 Discretionary Access Control

Windows NT implements a DAC policy using DACLs. Windows NT's philosophy of DAC and DACL assignment are described in this section.

5.2.1.1 DAC Philosophy

Windows NT implements a DAC policy between all protected objects and subjects. DAC provides a means of restricting access to objects based on the identity of a subject and/or the identity of groups (i.e., a collection of users) to which the subject belongs. It is at the discretion of the owner of an object, or any user who has WriteDac access to that object to define the access restrictions to that object. Windows NT uses DACLs as the mechanism to enforce DAC. As stated in Section "Access Control List Data Structure," on page 72 each entry in the DACL (an ACE) can specify an access to grant or deny to a particular user or group. There

are many different types of objects and each one has a set of defined access rights which can be allowed or denied in the DACL. Each of these specific access rights and the access rights which apply to all object types are discussed in detail in Section "Object Type-Specific Security Attributes," on page 143.

The enforcement of DAC the authority of object ownership, DAC revocation, and privileges which override the DACL are each described in this section.

5.2.1.1.1 DAC Enforcement The SRM validates a requested access to an object by a subject. It performs this validation by comparing the object's DACL against the identity(s) listed in the requesting subject's token. This validation routine is described in detail in Section "General Access Validation Algorithm," on page 76. As stated in Section "Object Types," on page 140, there are two categories of objects: executive and server objects.

For executive objects, a handle must be obtained by a subject before that subject can access that object. Handles are described in detail in Section "Handles and Handle Tables," on page 61. The ways to obtain a handle to an object are: object open, object create, inheritance, handle duplication, and special interfaces that allow for handle creation (NTCurrentProcess, NTCurrentThread, NTOpenProcessToken, NTOpenThreadToken). Each one of these results in handle creation and each one, with the exception of handle inheritance, results in a call to the SRM for access validation. The Object Manager creates handles for all executive objects. However, before the Object Manager releases a handle to a subject requesting access to objects, it calls the SRM for access validation. The Object Manager then stores the granted access mask, which it receives from the SRM (if the requested access right is valid), in the handle and releases the handle to the subject. All subsequent access requests to that object by that subject are then checked against the granted access mask stored in the handle. If an access right to an object is requested by a subject and that access right is not included in the granted access mask, then that access request is denied. Handle inheritance allows a child process to get copies of designated handles from its parent process. This is implemented by an inheritance bit in each of the handles of the parent process, which specifies the inheritance designation of the handle.

For server objects, a subject must open an object before any access to it is allowed. Each server calls the SRM for access validation before it opens an object. The SRM returns the granted access mask to the server and the server stores this mask in a type-specific structure (handle). This structure is used to restrict subsequent access requests by that subject to the object. As in the executive, if a different access is requested by a subject which is not in the granted access mask for that object, the request fails.

5.2.1.1.2 Object Ownership The owner of an object is specified in the security descriptor of that object as shown in Table 4.5, on page 75. The owner of an object is always allowed ReadControl access and WriteDac access regardless of what is specified in the DACL. WriteOwner is a standard access right which allows the owner of an object to be changed. The TakeOwnership privilege allows the same access as WriteOwner to all objects, without being granted the WriteOwner access in the DACL.

These access rights are described in Section "Overview of Object Access Rights," on page 142.

5.2.1.1.3 DAC Revocation A DACL can be changed on an object to which a subject currently has a handle. A change in an object's DACL will not impact subjects which already have a handle to this object. Access to objects is not immediately revoked; however, a change to an object's DACL will become effective upon future attempts to open that object.

5.2.1.1.4 DAC Related Privileges There are privileges which allow exception to what is specified in the DACL. Privileges are described in Section "Privileges," on page 151. If the primary or impersonation token of the process requesting access has any of these privileges enabled, the DACL is overridden. The privileges which allow exceptions to the DACL are listed below.

- TakeOwnership
- Security
- Backup
- Restore
- Debug
- ChangeNotify

Of the privileges listed, the SRM only knows about the TakeOwnership privilege (in the DAC related context) as described in step 1 of the routine in Section "General Access Validation Algorithm," on page 76. TakeOwnership privilege grants WriteOwner access to an object.

The other privileges listed above allowing DACL exception are enforced by other responsible subsystems which call the SRM privilege test routine (described in Section "Privilege Testing," on page 77). Backup and Restore privileges are enforced by the Configuration Manager and NTFS subsystems, and the debug privilege is enforced by the Process Manager subsystem. The ChangeNotify privilege provides the traverse access on directories. This privileges is given, by default, to all users and is not considered security relevant. The Security privilege provides several abilities. The SRM does know about its use to provide access to the SACL, and as stated in step 1 of the routine in Section "General Access Validation Algorithm," on page 76, enforces its privilege check. However, the Security privilege also provides access to the security log, overriding the DACL to the security log. The Event Logger is responsible for enforcing the Security privilege in this context.

5.2.1.2 Discretionary Access Control Lists

This section described the initial assignment of the DACL, the default DACL, and the difference between an empty DACL and an absent DACL.

When an object is created there are several ways for a DACL to be assigned. The DACL can be explicitly provided, can be a default DACL, or can be inherited. The explicitly provided method is when parameters defining the DACL are passed into the object create system call. Assignment of the DACL by default and via inheritance are described in this section.

5.2.1.2.1 ACL Inheritance There are two classifications of objects: container and noncontainer. Container objects (e.g., printer) can logically contain other objects and noncontainer objects (e.g., job) may not contain other objects. Objects within container objects may inherit the ACL from the container object. Each ACE within the ACL of a container object can be inherited by sub-objects of that container object. There are control flags which are only valid for container objects in each ACE which specify if and how that ACE can be inherited. Note that a sub-object can be either a container or non-container object. The inheritance control flags are listed in Table 4.4, on page 74, and again below.

- ObjectInherit - inherited by sub-objects of the container.

- ContainerInherit - inherited by sub-containers of the container.
- NoPropagateInherit - inheritance control flags are not propagated in ACEs inherited by sub-containers.
- InheritOnly - ACE is not to be used in validating access attempts to the corresponding object, but will be applied to sub-objects as they are created.

The inheritance control flags of the container object and the type of the sub-object (container or non-container) determine if and how ACEs are inherited. The policy is described in this section for both container and non-container sub-objects.

Container Sub-Object Inheritance Algorithm:

For each ACE in the object's ACL:

1. If the ContainerInherit flag in the container object's ACE is not set, then go to step 2. Otherwise:
 (a) A copy of the ACE is added to the end of the inherit ACL and the following actions performed on it.
 - The InheritOnly flag in the inherited ACE is cleared.
 - The NoPropagateInherit flag in the inherited ACE is copied from the container object's ACE.
 - The ObjectInherit and ContainerInherit flags in the inherited ACE are set according to the value of the NoPropagateInherit flag. If the NoPropagateInherit flag is cleared, then the ObjectInherit and ContainerInherit flag values are copied to the inherited ACE from the container object's ACE. Otherwise, the ObjectInherit and ContainerInherit flags in the inherited ACE are cleared.
 - If the ACE's access mask has any generic access flags set, then map the indicated generic rights to the standard and specific rights of the sub-object type.
 (b) If the container object's ACE is marked both ContainerInherit and InheritOnly, but not the NoPropagateInherit, then another copy of the ACE is added to the end of the inherit ACL without change.
2. If the ContainerInherit flag is not set and the NoPropagateInherit flag is not set, then the ObjectInherit and InheritOnly flags are checked. If both of these flags are set, then the ACE is added to the end of the inherit ACL.
3. Otherwise, the ACE is not inherited.

Non-container Sub-Object Inheritance Algorithm:

1. If the ObjectInherit flag is set, then the ACE is inherited and the following rules apply.
 - The InheritOnly flag in the inherited ACE is cleared.
 - The NoPropagateInherit, ObjectInherit and ContainerInherit flags in the inherited ACE are cleared.
 - If the ACE's access mask has generic access flags set, then map the indicated generic rights to the standard and specific rights of the sub-object type.
2. Otherwise, the ACE is not inherited.

5.2.1.2.2 Default Object Security All named executive objects and server objects must have a DACL. Unnamed executive object need not have a DACL with the exception of process, thread, and token object types. Named and unnamed objects are described in Section "Discretionary Access Control List Defaults," on page 142. The default methods described in this section apply to those objects which must have a DACL. When a DACL is not explicitly provided or obtained via inheritance, it will be assigned by default. Another DAC related security attribute, the owner attribute, will also be defaulted to if not provided explicitly. The security attributes of an object are stored in the security descriptor as described in Section "Security Descriptor Data Structure," on page 75. A field in the security descriptor points to the object's DACL. When a new object is created, the values assigned to the object's security descriptor are based on the following information: the user creating the object, any explicit values provided by the creator, any default values provided in the creating process' token, and any inheritable DACL information according to the rules stated in Section "ACL Inheritance," on page 146.

The following rules state how each DAC related security attribute is assigned to newly created named objects. The order of steps indicates the order the search for a value is performed. The search ends when the first value is found.

Owner:

1. An explicit or default value in the security descriptor, if provided.
2. A default value in the creating process' token, if provided.
3. The creator's SID provided in the creating process's token.

DACL:

1. The explicit DACL in the security descriptor, if provided.
2. The inheritable ACEs in the DACL of the container object which contains this object, if any. (As defined in Section "ACL Inheritance," on page 146).
3. The default DACL in the creating process' token.
4. No DACL is assigned, granting unconditional access to everyone. This cannot happen in the evaluated configuration. In the evaluated configuration, by default, every user is assigned a default DACL in the token of the first process created for them at logon. The default DACL is for the SYSTEM and the owner to get all access (see Section "LSA and Authentication," on page 111).

5.2.1.2.3 Absent vs Empty DACL There is a difference between the protection of an object that does not have a DACL (absent DACL) and an object that has a DACL with no ACEs in it (empty DACL). An empty DACL indicates that access is denied. An absent DACL indicates there is no protection assigned to the object, and unconditional access is granted.

5.2.2 Identification & Authentication

Before a user is allowed to do anything on a Windows NT system, the user must logon to the system and be authenticated. To logon to the system, the user first executes the Secure Attention Sequence (SAS),

resulting in a dialog box prompting for username and password. The user then types the username and corresponding password to logon to the system (see Section "Users and Groups," on page 137).

5.2.2.1 Identification

In Windows NT, every user or group account is identified uniquely by a Security Identifier (SID), a concatenation of several numerical values. SIDs were designed to have the following properties:

- Hierarchical - SIDs identify a chain of authorities and subauthorities, leading to the top-level identifier authority who generated the SID.
- Variable Length - Each authority and subauthority in the chain is represented in the SID by its own component. Thus, the more authorities included in this chain, the longer the SID will be. The maximum number of subauthorities is fifteen, although this is loosely enforced.
- Unique in Space & Time - No two SIDs of the same value will be generated.

As the system was implemented the hierarchy of sub-authorities became meaningless, and uniqueness is only guaranteed on a single system. SIDs are either hard-coded completely or contain some hard-coded fields together with pseudo-random numbers. However, the chance of two SIDs on different machines being the same is vanishingly small, as will be clear from how each is generated.

The fields defined in the SID data structure are shown in Table 5.3.

Field	Description
Revision	A 4-bit field containing the revision number of the SID data structure. By default, the current value of this field is one.
Reserved	A 4-bit field reserved for future use, it is currently set to zero and not displayed by the Event Viewer.
SubAuthorityCount	An integer value indicating the number of SubAuthority[] array entries included in the SID. This value is not displayed by the Event Viewer.
IdentifierAuthority	A 48-bit, hard-coded value. Its meaning is defined in the Win32 API
SubAuthority[]	An array of 32-bit relative IDs. It contains up to five numbers. The first is hard-coded. Depending on the value of the first, either the next number is hard-coded also or the next three are randomly generated. The fifth, if present, is incremented from one of several hard-coded starting values.

Table 5.3. Fields in the SID Data Structure

A typical SID is displayed by the system as follows:

S-1-5-21-148055195-1071131983-285021542-500

This number string is interpreted as follows:

'S': This is really not a part of SID. This letter is added to the beginning of the SID string for display purposes only to make it easier for one to recognize this string as a SID.

This means the actual SID stored in the computer is:

1-5-21-148055195-1071131983-285021542-500

where

- 1 represents Revision Field.
- 5 represents the Identifier Authority. This is a well-known Identifier Authority, namely, Windows NT Authority (hard-coded value of 5).
- 21 represents SubAuthority[0] and is also a hard-coded value that indicates the next three 32-bit values are randomly generated based on three different times during the installation process.
- 148055195 represents SubAuthority[1].
- 1071131983 represents SubAuthority[2].
- 285021542 represents SubAuthority[3].
- 500 represents SubAuthority[4] and is also hard-coded as the mandatory Administrator account. SubAuthority[4] for user accounts are incrementally generated beginning at the number 1000 and are never reused. This guarantees their uniqueness on a single system. Accounts belonging to privileged groups are similarly incremented from a hard-coded value identifying the privileged group. For example, SubAuthority[4] for a second administrator in the built-in Administrators group would be 501.

There are some well-known SIDs in Windows NT. The purpose of well-known SIDs is so applications can refer to them in their code with determinate results. These well-known SIDs along with their corresponding hard-coded values are given in Table 5.4.

Well-Known SIDs	**Corresponding Hard-coded SID Values**
Null SID	S-1-0-0
World	S-1-1-0
Local	S-1-2-0
Creator Owner ID	S-1-3-0
Creator Group ID	S-1-3-1
Non-Unique IDs	S-1-4
NT Authority	S-1-5

Table 5.4. Well-Known Hard-coded SIDs in Windows NT

For users with accounts that are not built-in to the system like Guest or Administrator, Windows NT allocates three 32-bit values during installation. These three longwords are derived from three pseudo-random numbers generated at different times during installation. These numbers are used to make the chance of SIDs on different machines being the same extremely small. These three numbers are used as the second, third, and fourth subauthorities. The fifth subauthority uniquely identifies a user on a single system. All subauthority values are called Relative IDs (RIDs).

Built-in groups like Administrators and Power Users have hard-coded SIDs. For example, the SID for Administrators is S-1-5-32-544. The subauthority value 32 indicates this is a built-in group.

5.2.2.2 Authentication

The username and the password have to be authenticated before the user is allowed to logon to the system. The username, associated SID, and encrypted password are managed by SAM, stored in the Registry, and protected by DAC. For more details on authentication mechanism, refer to the following: Section "WinLogon," on page 115; Section "Local Security Authority," on page 109; and Section "Security Accounts Manager," on page 112.

In Windows NT users may control their passwords to the extent allowed by system administrators. The password management policy can be set locally by the administrator. For each user, an administrator can do the following:

- Set the minimum password length
- Set password expiration date
- Maintain a history of previously used passwords
- Create and maintain a dictionary for screening of proposed passwords

These constraints are enforced by SAM.

5.2.3 Privileges

Privileges are stored in a process' token and allow a thread to take exception to an object's DACL or utilize restricted services. A thread is free to take advantage of those privileges listed in its token at any time. A process' privilege(s) is based on the user account on whose behalf the process is running and the privileges assigned to that user account. The privileges and the abilities or services they provide are listed in Table 5.5. The privileges that allow exceptions to the DACL are listed and further described in Section "DAC Related Privileges," on page 146.

The SYSTEM token is a special token in that processes associated with the SYSTEM token are said to be runing under the SYSTEM context. The SYSTEM token has a predefined set of privileges listed in it. Creating a SYSTEM token is an SRM function which can only be used by the executive. The privileges defined in the SYSTEM token and whether they are enabled or disabled by default is given in Table 4.9, on page 107.

5.2.3.1 Privilege Assignment

The Local Security Policy database in the LSA identifies which privileges are defined in the system and which users and local groups are assigned which privileges. During the logon process, the SAM retrieves the user's SID and the SID(s) of any local group(s) the user belongs to from its database. Using the SAM provided user and local group SIDs, the LSA collects what privileges these user and local groups have from its database and provides this information during logon. The user's SID, local group SID(s), and privileges are stored in the token of the process created for the user. More detail on logon is provided in Section "LSA and Authentication," on page 111. Once a user has logged on and the token is created, privileges cannot be added or removed from the token. Privileges can be added or removed from association with a user or group SID in the LSA's database at any time by a system administrator. However, if a change is made to

the privilege(s) of a logged on user in the LSA's database, this change will not take effect until the user logs on again and a new token is created.

5.2.3.2 Privilege Enforcement and Auditing

Each privilege allows a particular ability or service. The process requesting a privileged ability or service must have the associated privilege in its token and the process must enable the privilege (if it is not already enabled) before attempting to use the associated privileged abilities or services. Once privileges are initially assigned to a token (see Section "Privilege Assignment," on page 151, they cannot be removed nor can additional privileges be added. However, a process can disable and enable the assigned privileges using services provided by the SRM (see Section "Token Services," on page 70).

Privileged abilities and services are each implemented in the TCB. The TCB subsystem which implements the privileged ability or service checks to verify that the requesting process' token has the associated privilege enabled for the particular ability or service. The TCB subsystem accomplishes privilege verification by utilizing the services provided by the SRM (see Section "Privilege Testing," on page 77). The TCB subsystem responsible for implementing a particular ability or service also utilizes the SRM services provided to audit the use of privileges (see Section "Privilege Use Audit," on page 81).

Therefore, the TCB subsystem which implements and audits the use of the particular ability or service associated with a privilege is referred to as the enforcer of the privilege. The enforcer of each privilege is listed in Table 5.5.

5.2.3.3 Privilege Attributes

In the token, privileges have attributes associated with them. These attributes are listed below.

- Enabled - if set, the privilege is currently enabled and will be included in the privilege test routine described in Section "Privilege Testing," on page 77
- EnabledByDefault - if set, the privilege is enabled by default. If not set, the associated process must enable the privilege before it is used.
- UsedForAccess - if set, the privilege was needed for a particular access. This bit is set by the privilege test routine and used by the SRM in the audit routine of privilege use.

5.2.3.4 Privileged Users and Local Groups

There are predefined users and local groups that have privileges assigned to them. There is an Administrator account which is a user account that is a member of the Administrators group and therefore runs with all privileges associated with the Administrators group. This built-in account can be renamed but cannot be deleted. It also cannot be removed from the Administrators group (which is also built-in).

There are several built-in local group accounts that have privileges associated with them. These privileges can be granted or removed from these built-in local group accounts.

The built-in groups are: Administrators, Backup Operators, Power Users, Users, Guests, and Replicator. The Administrators built-in local group is the most privilege built-in local group. The built-in privileged

local groups are listed along with their associated privileges for each group in Table 5.6. The Replicator Group supports consistency between multiple copies of the same files or directories and is only meaningful in a network configuration and is therefore not applicable to the evaluated configuration. It is therefore eliminated from Table 5.6. Other abilities associated with each one of these built-in accounts are listed in Table 5.1, on page 139.

5.2.4 Object Reuse

Windows NT provides for secure object reuse in three different ways. One or more of the following methods is used for each object protected by the Windows NT TCB:

- All fields in the generic object header and the object type-specific body are initialized to appropriate new values
- Memory allocated for objects is zeroed
- Exactly what is most recently written can be read and no more

Some objects like File objects must be persistent and require backing store by disk. Files on disk are subject to security access checks based on the DACLs of the corresponding File objects. Access to a file is also controlled by the use of a high water mark and an end-of-file marker. The high water mark indicates the largest size of the file prior to current use. A new process with access rights to the corresponding File object can move the end-of-file marker past the high water mark but the file system immediately zeroes the disk or tape space between the old high water mark and where the end-of-file marker has moved. No process can read past the end-of-file marker.

Other objects use backing store to minimize the amount of dynamic memory needed at any one time. In particular, the Configuration Registry is backed on disk. Data for Key objects and their associated key values are stored there. When a Key object is created, it is entirely initialized. When a key value is created, it is set to exactly the value specified; the next read of that key value will return exactly that value and nothing else, even though key values, unlike Key objects, are different in size.

The first two subsections will detail, for each subsystem in the executive and each protected server in the user-mode portion of the Windows NT TCB, how that subsystem or protected server handles object reuse for the objects that it exports to user-mode processes. The third subsection will discuss hardware reuse.

5.2.4.1 Executive Subsystem Objects

All executive objects are created by the Object Manager and reside in a single kernel-mode address space. The Object Manager initializes all fields of the generic object header for all object types before passing newly created objects to the requesting executive subsystem. Thus, previously used memory for a newly created object header is overwritten completely. It is the responsibility of the executive subsystem that called for the object's creation to initialize the object type-specific body of the object. All executive subsystems are part of the TCB, so each is trusted to clear old data from previous object use of kernel-mode memory. Executive objects cannot be directly accessed by a user-mode process. User-mode processes must request a handle to the executive object from the Object Manager, and all attempts to access the object using the handle will

Privilege Name	Description	Enforcer
CreateToken	To create a token	SRM
AssignPrimaryToken	To assign the primary token of a process	Process Manager
LockMemory	To lock physical pages in memory	Memory Manager
IncreaseQuota	To increase the quota assigned to a process	Memory Manager
IncreaseBasePriority	To increase the base priority of a process	Process Manager
TCB	Identifies its holder as part of the TCB	Win32, LSA, Configuration Manager, Executive Object Services
Security	Identifies the holder as a security operator. Allows access to SACLs and to the audit log	Win32, Object Manager, SRM, NTFS, Event Logger
TakeOwnership	To take ownership of an object without being granted discretionary access to do so. The owner value can only be set to a SID of an account in the token of the user with this privilege.	SRM
Audit	To request audit generation	SRM
SystemTime	To modify the system time	Executive Object Services
CreatePagefile	To create a paging file	Memory Manager
CreatePermanent	To create a permanent object	Object Manager
Backup	To perform backup operations	Configuration Manager, NTFS, I/O subsystem
Restore	To perform restore operations	Configuration Manager, I/O subsystem, NTFS
LoadDriver	To load or unload a device driver	I/O subsystem
Shutdown	Shut down the system	Win32, Session Manager, WinLogon, Executive Object Services
Debug	Allows for the opening of any process. The TFM will instruct the administrator not to assign this privilege to any users.	Process Manager
SystemEnvironment	Required on the Alpha to change the system's non-volatile ram (boot configuration information)	Executive Object Services
SystemProfile	To gather profiling information for the entire system	Executive Object Services
ChangeNotify	Allows traverse access on directories. This privilege is not security relevant, it will be granted to all users by default.	I/O
MachineAccount	Only relevant in a network configuration and is therefore not applicable to the evaluated configuration.	SAM
RemoteShutdown	Allows for the ability to remotely shutdown the system. This privilege is not applicable in the evaluated configuration since it is only relevant in a network configuration.	WinLogon

Table 5.5. Privileges

Privileges	Admin	PowerUsers	BackupOp	Users	Guest
TakeOwnership	•				
Security	•				
Change System Time	•	•			
Shutdown System	•	•	•	•	•
Backup/Restore	•		•		

Table 5.6. Windows NT Privileged Built-in Local Groups

necessarily be subject to security checks. The following subsections describe how object reuse is handled for the objects exported to user-mode processes by each of the Windows NT executive subsystems. Reuse of objects that are not exported (e.g., queues) is not described.

5.2.4.1.1 Executive Object Services The Windows NT Executive Object Services subsystem exports these different microkernel objects: IoCompletion Port, Event, Event Pair, Mutant, Semaphore, Profile, and Timer. All fields of the body of IoCompletion Port, Event, Event Pair, Mutant, and Semaphore objects are initialized by Executive Object Services. All but two fields of the body of a Profile object are initialized by Executive Object Services after the object is created. The remaining fields are initialized when a Profile object (used for measuring execution time) is started. All but one field of the body of a Timer object are initialized by Executive Object Services after the object is created. The remaining field is initialized when a call is made to set the Timer object.

5.2.4.1.2 I/O Manager The I/O manager manages File, Named Pipe, and Mailslot objects. This subsystem uses the generic File object to represent all three object types plus devices and directories. The meaning for the fields in the generic header are different depending on which object type is represented by the File object. For each of these object types, the I/O subsystem ensures that the object body is zeroed after its creation and then its fields are initialized with information appropriate for the object type. Named pipe and Mailslot objects are represented by memory in the executive's address space. Hence, access checks are performed when either reads or writes are requested by users. In addition, the I/O subsystem will not allow a user to read more than what has been written, so if this memory space contains data from a previous use, it cannot be read.

5.2.4.1.3 Object Manager The Object Manager manages Directory and Symbolic Link objects. For a Directory object the Object Manager zeroes the object body after it is created. All fields of Symbolic Link objects are initialized by the Object Manager with information that is passed by the user-mode process requesting the Symbolic Link object creation.

5.2.4.1.4 Security Reference Monitor The Security Reference Monitor manages Token objects. All fields in the body of a Token object are initialized by the Security Reference Monitor after the object's creation.

5.2.4.1.5 Process Manager The Process Manager manages Process and Thread objects. After the creation of a Process object by the Object Manager, the Process Manager zeroes all fields of the Process object body. Similarly, all fields of the body of a Thread object are zeroed.

5.2.4.1.6 Memory Manager The Memory Manager manages Section objects, as well as the allocation of memory in general.

The Memory Manager initializes the Section object body. A Section object is initialized in one of three ways, depending upon its intended use. If a Section object is to be used for paging, the Section object body is initialized with the process image. If a Section object is to be used for mapping a file into a process' address space, then the Section object body is initialized with the file's contents. If a Section object is to be used for any other purpose, its body is zeroed after creation.

5.2.4.1.7 LPC Subsystem The LPC subsystem manages Port objects. The body of a Port object is first zeroed by the LPC subsystem, then initialized with appropriate information.

5.2.4.1.8 Configuration Manager The Configuration Manager maintains Key objects. Associated with a Key object are zero or more key values. The body of a Key object is completely initialized by the Configuration Manager after its creation. A Key value is initialized with exactly the value supplied by the thread which calls for the key value to be created. A thread requesting a read of a key value will receive exactly the value most recently written.

5.2.4.2 Protected Server Objects

Protected Server objects are allocated from heap memory in a user-mode process address space. This memory is zeroed by the Memory Manager before it is mapped into the process' address space. Thus, no information from a previous process that used the same physical pages will remain in the new process' address space. In addition, if the Memory Manager extends the heap, it will zero the extension. The following subsections describe how object reuse is handled for the objects managed by each of the Windows NT user-mode protected servers.

5.2.4.2.1 Win32 The Win32 Server manages Desktop and WindowStation objects. There is one WindowStation object associated with the keyboard, mouse, and screen. It is initialized completely during system startup and is never reused. Other WindowStation objects are created and initialized during service logons. New services requiring service logons can only be installed by members of the Administrators group. Upon allocation of memory for a Desktop object, all data is zeroed by the Memory Manager. The Win32 Server has four shared sections. The default desktop shared section is deallocated when a user logs off. The screensaver desktop shared section is allocated when a screensaver program starts and is deallocated when the screensaver program ends. The WinLogon desktop shared section persists across logons, but is accessible only within the TCB. The handle table shared section is not deallocated but the handles are cleared between logons. Even though the handle table is shared, it contains only integers to reference physical locations of objects; it does not contain any user data, which is protected by the Server.

5.2.4.2.2 Win32 Spooler The Win32 Spooler manages Print Server, Printer, and Job objects. There is no old information in these objects because the memory used by these objects was zeroed before being allocated for reuse.

5.2.4.2.3 Local Security Authority The Local Security Authority manages the Policy object, Trusted Domain objects, Account objects, and Secret objects. There is only a single instance of the Policy object in Windows NT. It is created during system startup, all of its fields are initialized, and it cannot be deleted. All fields in the body of Trusted Domain, Account, and Secret objects are initialized after their creation.

5.2.4.2.4 Security Account Manager The Security Account Manager manages the SAM Server object, Domain objects, User Account objects, Group Account objects, and Alias Account objects. There is only one SAM Server object per system, and both its fixed and variable parts are completely initialized on startup. After startup it cannot be destroyed or recreated. Domain, User Account, Group Account, and Alias Account objects also consist of a fixed part and a variable part that are completed initialized when created.

5.2.4.2.5 Service Controller The Service Controller is responsible for Service objects. When a new Service Object is created, the Service Controller initializes all fields in the body to zero.

5.2.4.2.6 Event Logger The Event Logger manages Log objects. The only Log object that is security relevant is the security Log object, and it is never reused. It also can only be read by members of the Administrators group, who are all trusted.

5.2.4.3 Reuse of Hardware

Secure hardware reuse, including reuse of integer and floating point registers, VGA memory, and special purpose registers like the processor status register, is assured. Each storage location is under control of either the Executive part of the TCB, WinLogon, or Win32. These storage locations are either cleared when one interactive user logs out, or are cleared when they are reallocated.

When a thread is started, Windows NT initializes all processor registers to a known state. When context is switched, processor state is saved, and all registers are reinitialized.

The one exception to this is floating point registers. When context is switched, floating point state is not immediately saved. Instead, the processor is configured to fault if a floating point instruction is issued in the new context. If that occurs, the state of the floating point registers is saved from the previous context, and the floating point registers are initialized. A thread cannot execute a floating point instruction unless its floating point context is active.

Data from an interactive user session in the VGA memory is cleared during logoff. No data that was displayed by a previous interactive user can be seen after he logs off. Although VGA memory can be accessed directly under a virtual DOS machine, this would be the single interactive user's own data, and thus not a security problem.

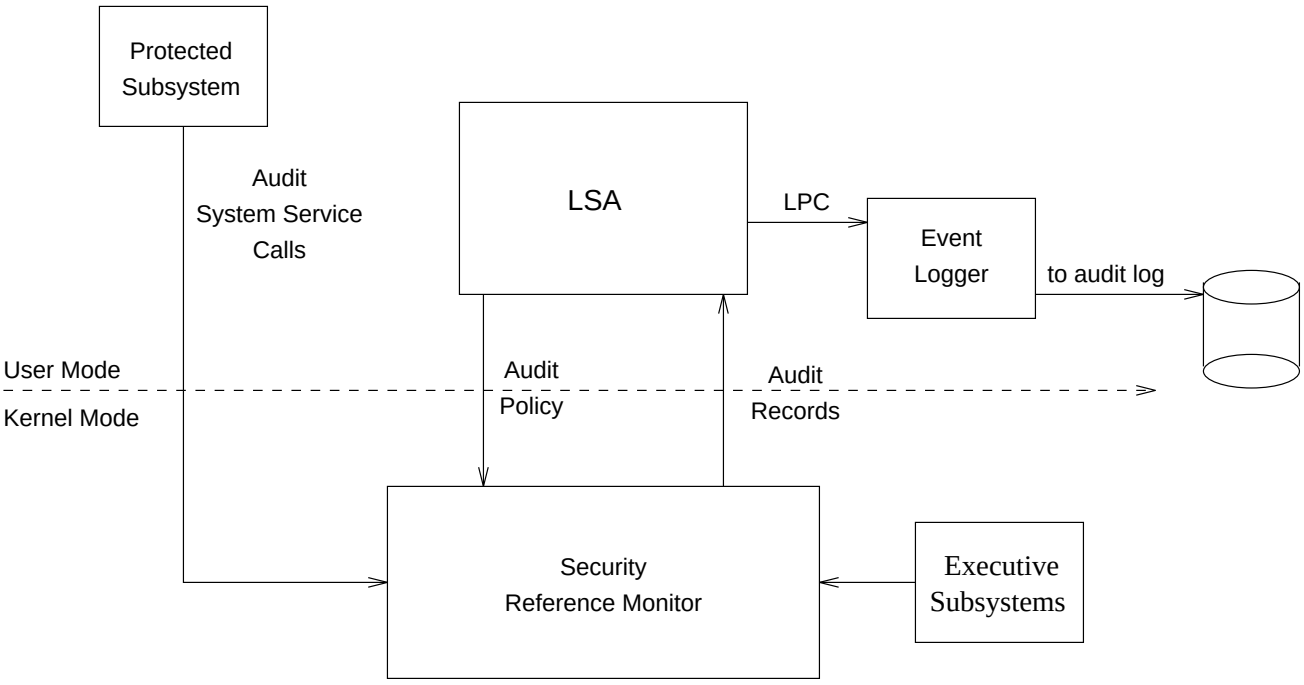


Figure 5.1. Audit Flow

With regard to I/O hardware object reuse, including drive controller status registers, keyboard buffers, and data caches, among other objects, access to these objects require administrative or system privileges, and attempts to access these objects without privilege results in a general exception, and subsequent denial of access. Hence, no non-privileged user can examine data or status information remaining from a previous user.

With regard to printer buffer reuse, the printer identified in the evaluated configuration is a Hewlett Packard LaserJet IV. Printer language code is sent before each print job, that essentially clears printer memory of any user data and programs, and prevents any user from examining data remaining from a previous user's print job.

5.2.5 Audit Mechanism

Several components contribute to the audit mechanism: the LSA, the SRM, protected servers and executive subsystems, the Event Logger and the Event Viewer.

The role of each of these components is described in more detail in this section and a description of the contents of the audit log is also provided. The flow of audit traffic is depicted in Figure 5.1.

5.2.5.1 Audit Policy Definition

The audit policy is defined by the parameters that the LSA provides to the SRM. This policy is passed to the SRM via LPC and consist of two parts. The first part is a boolean value indicating whether auditing is enabled. The second part is only valid if auditing was enabled in the first part and is in the form of an array of event categories with each category having flags associated with it. These flags indicate if successful, if unsuccessful, if both successful and unsuccessful occurrences of events in each category should be audited. The event categories are: detailed tracking, system, logon/logoff, object access, privilege use, policy changes, and account management. The event types that each event category includes is listed in Table 5.7.

5.2.5.1.1 Server and Subsystem Audit Specification The SRM constructs the audit records for the detailed tracking, system, object access, and privilege use categories. Specifying these event categories in the audit policy only makes these events auditable. In other words, audit records for these categories are not automatically generated. To actually cause an audit record to be generated, the responsible subsystem or server must make a request to the SRM that an audit record be generated upon each occurrence of the event type. The SRM provides audit system services to both kernel-mode and user-mode for the subsystems and servers to request audit generation. Request to generate audit records from within the executive are always honored by the SRM. However, servers using the SRM audit services must have the Audit privilege.

The SRM checks for the Audit privilege in the primary token of the process requesting the generation of an audit record. The use of the Audit privilege itself is not audited. The ability to have an audit record inserted into the audit log (security log) is not security relevant because it does not provide direct access to the security log and the source of the audit record is always recorded.

5.2.5.2 Event Logging and Viewing

The Event Logger writes audit records to the security log and the Event Viewer is used by the administrator to view the security log. These components are further described in this section.

5.2.5.2.1 Event Logger and Audit (see Section "Event Logger," on page 117). The Event Logger (see Section "Event Logger," on page 117 is used to write the audit records into a log file and provides services for a process to enter events in a log and for a viewer to read events in the log for display. The LSA uses the Event Logger services to write audit records to the security log. The LSA communicates with the Event Logger over a named pipe.

To manage the security log the Configuration Manager services must be used. Managing the security log entails the following:

- Specify log files based on the module that is logging, which allows for the ability to specify the same log file for many modules logging events.
- Specify the maximum size of the security log and what to do when this size is reached (e.g., wrap or shutdown the system).
- Specify a retention period, indicating the age at which audit records will be overwritten

Category	Event Type	Constructor
System	System Restart	SRM
	System Shutdown	SRM
	Authentication Package Loaded	LSA
	Registered Logon Process	LSA
	AuditLog Cleared	SRM
	#Audits Discarded (due to full internal queues)	SRM
Logon/Logoff	Logon Successful	LSA
	Unknown User or Password	LSA
	Time Restricted Logon Failure	LSA
	Account Disabled	LSA
	Account Expired	LSA
	Invalid Workstation	LSA
	LogonType Restricted	LSA
	Password Expired	LSA
	FailedLogon	LSA
	Logoff	LSA
Object Access	Open Object	SRM
	Close Handle	SRM
Privilege Use	Assign Special Privileges	SRM
	Privileged Service	SRM
	Privileged Object Access	SRM
Detailed Tracking	Process Created	SRM
	Process Exit	SRM
	Duplicate Handle	SRM
	Indirect Reference	SRM
Policy Changes	Privilege Assigned	LSA
	Privilege Removed	LSA
	Audit Policy Change	LSA
Account Management	Domain Changed	LSA
	User Changed	LSA
	User Created	LSA
	User Deleted	LSA
	Global Group Member Removed	LSA
	Global Group Member Added	LSA
	Local Group Changed	LSA
	Local Group Created	LSA
	Local Group Member Removed	LSA
	Local Group Member Added	LSA
	Local Group Deleted	LSA

Table 5.7. Audit Event Types

The event logging management information is stored in the configuration database in

HKEY_LOCAL_MACHINE/System/Current_Control_Set/Services/Eventlog.

5.2.5.2.2 Event Viewer and Audit The Event Viewer is the utility used for viewing the security log and generating reports. It uses the viewing services provided by the Event Logger to view the security log. The Event Viewer expects an event source module to be defined as part of the security log information in the configuration registry. Only one event source module needs to be defined since the LSA is the only source sending audit records to the Event Logger. Each event source module must define the following information in the configuration registry: event message file, category message file, and a parameter message file. The event message file contains the displayable strings for each audit message corresponding to the audit event types. It includes parameter substitution markers to be replaced at viewing time with strings. An event source module is shipped with the product for the security log. The category message file indicates the types of events supported and a category count. The parameter message file is used to provide object type-specific access names. When the event viewer is asked to display an audit record, it uses the event source module name and event ID from the record to retrieve a message string for that event. Each event type has an event ID associated with it. The Event Viewer formats individual parameter strings received in the event record and then formats the entire message string.

A user must have the Security privilege to use the Event Viewer to view the security log or be granted access via the DACL of the security log. The searching and filtering abilities of the Event Viewer are described in Section "Event Viewer," on page 129.

5.2.5.3 Audit Categories

5.2.5.3.1 System Events types in this category indicate events that affect the security of the entire system including: system rebooted, system is being shutdown, authentication package loaded, logon process registered, security log cleared, internal queues were full. The audit records for the event types indicating that the internal queues were full include the amount of records discarded. The SRM constructs the audit records for each of the event types in this category. The audit records for the event types indicating that the internal queues are full and that the security log was cleared are the only audit records that are generates automatically. The LSA does not control the generation of these event types as part of the audit policy.

5.2.5.3.2 Logon/Logoff Events types in this category indicate a successful or unsuccessful logon or logoff. These event types are: successful logon, attempted access using unknown user account or password, attempted access to a time restricted account, account disabled, account expired, logon type restricted, password expired, unsuccessful logon,[2] and logoff. All of these audit records are constructed by the LSA.

5.2.5.3.3 Object Access The event types in this category indicate access to objects including opening and closing an object. If this category is specified in the audit policy to the SRM then each occurrence of opening an object is subject to the evaluation of the object's SACL.

As described in Section "Access Control List Data Structure," on page 72 the SACL is part of an object's security descriptor representing the auditing control of access to that object. Each entry in the SACL

[2] logon rejected for some reason not covered by the other event types

contains: control flags, an access mask, and an SID. The control flags indicate whether audit records should be generated for successful access attempts, failed access attempts, or both. The access mask and SID indicate the particular access right(s) when requested by a particular user or group to that object which should be audited (according to the control flags). The SACL is evaluated only if the responsible subsystem or server uses the SRM services to audit each occurrence as described in Section "Server and Subsystem Audit Specification," on page 159. Depending on the SACL evaluation, an audit record will or will not be generated. This evaluation and audit generation is explained in Section "Object Access Audit," on page 80.

The SACL can be inherited as described in Section "ACL Inheritance," on page 146. This allows the administrator the ability to control auditing of objects as they are created. A tool (see Section "File Manager," on page 129) is available for the administrator to create and modify SACLs on directories and files and also to specify a SACL's inheritability.

Each audit record for object open event type contains the object type, object name, responsible subsystem, operation ID, process ID, primary and impersonation ID, granted access mask if successful, desired access mask if unsuccessful. The close handle audit record does not contain the object name. However, it does contain the handle ID and the process ID. These IDs can be used to find the open object audit record where the object name is provided. This is the name of the object which is now being closed.[3] The audit records for the event types in this category are constructed by the SRM.

For executive objects, creating a handle to an object is also audited. The audit record for handle creation is combined into the audit record for object open.

5.2.5.3.4 Privilege Use The event types in this category indicate events of successful and unsuccessful attempts to use privileges. Including this category in LSA's audit policy to SRM indicates that audit records should be generated for privileged object access and the use of privileged services. A description of privileged object access and privileged services is given in Section "Privilege Testing," on page 77. Each occurrence of each of these events must be audited using the SRM audit services as described in Section "Server and Subsystem Audit Specification," on page 159. SRM's auditing functionality of the use of privilege is described in Section "Privilege Use Audit," on page 81. There are some privileges to which specifying auditing the use of privilege in the audit policy does not apply. The individual use of these privileges is not auditable. These privileges are referred to as *special privileges* two of which are security relevant: BackupPrivilege and RestorePrivilege.[4] Even though the use of these special privileges is not auditable, including this category in the audit policy does make the assignment of them auditable. Audit records will be generated when the special privileges are assigned. Each audit record for these event types include the privilege name and privilege service, if applicable. Audit records for these event types are constructed by the SRM.

To audit the use of Backup and Restore a registry key must be set by the administrator. However, privilege use must be specified in the audit policy to SRM or else setting this key will not be effective.

5.2.5.3.5 Detailed Tracking The event types in this category provide detailed process tracking information. The events types indicate: process creation, process exit, handle duplication,[5] and an *indirect reference*. Process exit is important to track because process IDs are reused. An indirect reference occurs when the creation of one object causes an access check on another object. The only time this happens in

[3] This effort is to meet the requirement of auditing the name of objects deleted from the user's address space
[4] Special Privileges: ChangeNotify, Audit, CreateToken, AssignPrimaryToken, Backup, and Restore, Debug
[5] Duplication of protected server objects is not a supported concept. Handles of kernel objects can only be duplicated.

Windows NT is when connecting to an LPC port. When detailed tracking is specified, audit records for process creation and process exit events will be generated. Process IDs are included in audit records regardless of whether detailed tracking is specified. However, when detailed tracking is specified, audit records with process IDs can be correlated the process creation and process exit audit records to make it obvious to the administrator when process IDs are reused. Note that process IDs can only be displayed by audits whose generation was requested by the executive.

5.2.5.3.6 Policy Change The event types in this category indicate changes to the security policy. These event types are: adding and removing privileges to and from users, and changing the audit policy. When privileges are assigned or removed, the privilege name is included in the audit record. When the audit policy is changed, the new policy is provided in the audit record. There are no failed audit records for these events.

5.2.5.3.7 Account Management The event types in this category indicate changes to the SAM database. These event types are: a user account is created, changed or deleted; a user has been added or removed from a global group; a local group has been created, changed or deleted; and a user has been added or removed from a local group. There are no failed audit records for these events.

5.2.5.4 Audit Records

This section describes: what happens with audit records which are created during system startup before the audit mechanism is in place, the construction of audit records, identification and the content of audit records, and how to prevent the loss of audit records.

5.2.5.4.1 Startup Audit Records During system startup, the LSA starts before the Event Logger. The LSA keeps attempting to open the security log until it has a valid handle. If the attempt fails, it is assumed that the Event Logger has not started yet and the LSA maintains a queue of any audit records while it is waiting for the Event Logger to start. When the LSA gets a valid handle to the security log, it keeps it open permanently. At this point, if there are any audit records in its queue, the LSA will log each audit record in the queue and then go into normal auditing mode. This queue has a maximum length and audit records will be discarded after this length is reached.

5.2.5.5 Construction of Audit Records

The LSA and the SRM each construct audit records. The SRM constructs audit events for all the categories except logon/logoff, account management and policy changes. The audit records constructed by the SRM are put on an SRM internal queue to be sent to the LSA. The queue restrictions and how these records are sent to the LSA is described in Section "Audit Policy," on page 79. The LSA unpacks, expands, or converts data in the audit record as required. The LSA expands the logonID into a user name, domain name, and logonID by looking up the logonID in an LSA internal database. The LSA expands the access mask by converting bits to strings.

The logon/logoff, policy changes, and account management audit event records are constructed by the LSA. LSA provides routines which are called by SAM to generate audit records.

The LSA sends the audit records it constructs as well as the audit records passed to it from the SRM to the Event Logger over a named pipe. LSA and the Event Logger both use heap storage for these records and completely write over the storage they allocate for the audit record. When the buffer is released back to heap storage, it will be reinitialized. Only what is written to a named pipe can be read. The audit information is only kept in LSA long enough to update it, format it, and send it out to the Event Logger. The audit data is only kept in the Event Logger long enough to write it to disk. The security log is written to an NTFS file and is protected by a DACL.

5.2.5.5.1 Audit Record User Identification In each audit record a user SID will be provided to identify the responsible user account. This is sufficient when impersonation is not involved. However, when impersonation is involved the user SID may be impersonating another user SID. In this case, there is a primary SID and an impersonation SID which are both included in the requester's token. When a process is impersonating, both the SIDs are included in the audit record. They are called the primary SID and the client SID. When a log entry is displayed, the event viewer reads the SID in the audit log and calls the LSA to translate it into a domain/username pair. The event viewer then displays what LSA returns.

Handle IDs are included in many audits allowing audit record association assisting the administrator to determine the duration of time a particular object was open.

5.2.5.5.2 Audit Record Contents Each audit record includes the following header information:

- Time event was generated
- SID associated with the process causing event to be generated. If impersonating, this SID will be the impersonation SID. If not impersonating, this SID will be the primary SID. (Both the primary and impersonation SID are included in the body of the audit record when applicable).
- Name of component or module that submitted the event. (Will always be security for security audits)
- Event ID
- Event type: error, warning, success, information, success audit, failure audit.
- Event Category
- Computer

5.2.5.6 Audit Loss Prevention

There are situations in the system where audit records can be lost: when the SRM internal queues reach their high-water mark as described in Section "Audit Policy," on page 79, when the security log file becomes full, and when the internal queues of the LSA reach their limit as described in Section "Startup Audit Records," on page 163. The management options available on the security log file are described in Section "Event Logger and Audit," on page 159. To prevent audit loss, the wrap around option must not be chosen. A maximum size of the security log should be provided and an indication to halt the system when this maximum is reached must be specified. This indication will be in the form of a registry key. When set, it will indicate to the system that it is running in a mode in which audits may not be lost (crash on fail). When an audit record cannot be logged, this key will be checked and if set, the system will be halted without

automatic reboot. At this time auditing is still turned on, however, crash on fail is off. Therefore, only an administrator should be allowed to re-boot the system and is responsible for resolving the audit situation before performing any other auditable actions. Administrative intervention will be enforced here because of the machine's physical security.

The administrator will use the Event Viewer to choose the audit log options that sets the registry key to crash on fail. When this feature is chosen by the administrator the high-low water mark mechanism of the SRM internal queues is automatically disabled and the maximum length of the LSA internal audit queue is made invalid. The only limit of these internal audit queues becomes the size of the available memory.

This page intentionally left blank

Chapter 6

Assurances

Assurance that the security mechanisms work correctly to enforce the system defined security policy is a fundamental requirement of a trusted product. The level or degree of this assurance provides the basis by which users of the product can gauge the level of risk to the loss of their information that may be present when using the product. This level of assurance is important in product or system certification and accreditation activities.

The following sections describe the assurance features of Windows NT. A description of the mechanisms which ensure that the Windows NT TCB cannot be corrupted is provided, followed by a description of the supporting evidence (i.e., design documentation and specifications) for those mechanisms. The security testing performed by Microsoft will be described as well as Microsoft's procedures for ensuring the continued maintenance of the security features and product assurances (i.e., RAMP procedures).

6.1 System Architecture

This subsection describes the Windows NT mechanisms which prevent the tampering with or modification of the Windows NT TCB. References to other sections of this report where additional detail may be found are provided.

6.1.1 TCB Definition

Before describing how the Windows NT TCB is protected from modification and/or tampering, the following categorization of TCB subsystems is provided. Each subsystem of the Windows NT TCB falls into one of the following categories:

- Hardware and Firmware, as described in Section "Evaluated Processor Architectures," on page 11.
- The Windows NT Executive, as described in Section "Executive," on page 49.
- The Windows NT Non-Executive TCB, consisting of user-mode protected servers, as described in Section "Protected Servers," on page 106.
- Administrative Tools, described in Section "Administrator Tools," on page 128.

6.1.2 Process and Resource Isolation

Process isolation is achieved through the virtual memory mechanisms of the Windows NT Memory Manager executive subsystem, see Section "Memory Manager," on page 63. The Memory Manager provides each

process with a 4 GB address space which consists of shared and unshared portions. The upper 2 GB of each address space is reserved for the Windows NT Executive. The remainder is unique for each user-mode address space. Address space separation is achieved through the use of private, per process, page tables. The details of memory allocation, protection, and sharing were described in Section "Memory Manager," on page 63.

Executive resources are isolated by the Object Manager subsystem of the Windows NT Executive. The Object Manager uses the Security Reference Monitor to mediate all accesses to all executive mode objects. All other subsystems of Windows NT utilize the services provided by the Object Manager. No subsystem bypasses the Object Manager to directly export objects outside of the Windows NT Executive. A more detailed description of the services of the Object Manager was provided in Section "Object Manager," on page 58.

Resource isolation is achieved in Windows NT by requiring that all accesses to an executive object must be preceded by obtaining a handle to the object, specifying the allowed access modes for the process. Handles are stored in the handle table, which is maintained by the Object Manager and bound to a process by the Process Manager. The Object Manager enforces the security policy via calls to the Security Reference Monitor whenever object access requests are made.

The Non-Executive Protected Servers utilize the resources and services provided by the Windows NT Executive to export their own set of resources. These resources are maintained within the address space of the protected server or are stored in the Registry files. Enforcement of the system security policy for these non-executive resources is provided by the protected server itself. The protected servers maintain separation of client information while servicing the requests of multiple threads. Clients cannot access information belonging to others clients as their only access to the protected servers is through the IPC mechanism, thus clients may only receive what the protected server determines to be appropriate (i.e., as belonging to that client).

6.1.3 TCB Integrity

The Windows NT Executive is protected from modification and tampering by untrusted processes via the multi-state and/or address translation mechanisms of the underlying hardware platform(s).

The Windows NT Executive runs entirely in the hardware privileged mode, called kernel-mode. A context switch from user-mode to kernel-mode can only be made through the use of the mechanisms available at the TCB interface (i.e., hardware instructions, system service calls, and APIs), these are described further in Section "TCB Interfaces," on page 9.

Each Windows NT Non-Executive Protected Server is implemented as a separate process, therefore the integrity of these servers is maintained through the address space separation mechanisms described in Section "Virtual Memory Management," on page 63.

The files which contain the executable programs and data files which comprise the Windows NT TCB are protected on secondary storage via the Windows NT DAC mechanism.[1]

[1] This must be confirmed during testing.

6.2 Design Documentation

A large amount of documentation exists for Windows NT. This documentation is in the form of publicly available material such as *Inside Windows NT*, the five volumes of the *Win32 Programmers Reference*, and the three volume *Windows NT Resource Kit*, as well as Microsoft internal/proprietary documentation such as design and interface specifications for all Windows NT subsystems. A complete list of this documentation can be found in Section "Bibliography and References," on page 195.

6.3 Test Documentation

The test documentation is organized following the vendor's philosophy of testing. Test descriptions are divided into files that either describe tests for a protected server or the executive, a security mechanism, or an administrative tool. These files are called High Level Description (HLD) files.

In addition, the vendor supplied a Test Overview document. The first part of this document describes the purpose of each test suite, what each suite is meant to verify, and the file that contains a more detailed description of the test suite.

The second part of the Test Overview document contains three high-level matrices and several low-level matrices and tables. The first high-level matrix identifies a low-level matrix for each protected server and applicable security mechanism. The low-level matrix identifies the HLD file(s) and sections for each protected server API and security mechanism. The second high-level matrix points to a single section of the overview document that identifies detailed test documentation for each security-relevant executive API and security mechanism. The third high-level matrix identifies a low-level table for each administrative tool and applicable security mechanism. The low-level table contains a column for test category, a column for test description, and a column pointing to the HLD file that describes the test and security mechanism.

6.4 Security Testing

The vendor's philosophy of testing is three-pronged. It includes testing the security-relevant functionality of the TCB's protected servers, which execute in user mode, and the executive, which executes in kernel mode, through their respective APIs. This testing is extensive and generally treats each system component as a black box. In addition, the vendor's philosophy of testing is to focus separate test suites on major security-relevant mechanisms such as object access protection, object reuse, and the use of privileges. Finally, the vendor's philosophy of testing includes testing all security-relevant access to the TCB through the user interface of Windows NT's administrative tools.

There are two types of vendor tests, automatic and manual. In general, the security-relevant APIs of the protected servers and the executive are tested automatically by programs that exercise numerous calls to the APIs, each with different arguments or with different test configurations. These calls are described as variations in the vendor's test documentation. The results of these variations are logged in a file for later analysis. The log files contain assertions of what the test is meant to demonstrate and the success or failure of the test with regard to these assertions. These tests are grouped by component (protected server or executive) or by security-relevant mechanisms (e.g., object reuse).

6.5 System Integrity

The following system hardware integrity tests are included with or are available for the evaluated systems. Tests are described for processors and for systems.

The Intel x86 and DEC Alpha CPU integrity tests will be described first, followed by descriptions of integrity tests for the Compaq Proliant 2000 and 4000 and the DECpc AXP/150.

6.5.1 Intel Pentium Hardware Diagnostics

Processor security functionality tests are available from Microsoft to test the security-relevant CPU functions. These are intended to be periodic diagnostic tests which can be run by the system administrator to verify the proper operation of the processor's protection mechanisms. The following processor features are tested.

Protected mode can be initialized. Switches to protected mode and verifies that the correct values are set in the Primary Control Register, General Flag Register, the CPL and DPL in the Segment Selectors, the Local, Global, and Interrupt Descriptor registers, the Page Tables and Page Directory Base Register, and the RPL in the Operand Segment Selectors.

No privileged instruction can be executed in user mode. In user mode (Ring 3) each privileged instructions is attempted and the correct fault is generated for each. Four groups of instructions are tested: those that execute on both the i486 and Pentium, those that are i486 specific, those that are Pentium specific, and those that are I/O Privilege Level related.

Control transfers between user and kernel modes work correctly. This test switches between the two processor modes used by Windows NT and verifies that the correct privilege levels are active at each stage. Specifically, it tests:

- Ring 3 to Ring 0 transfers
- Ring 0 to Ring 3 transfers
- V86 mode to Ring 0 transfers
- Ring 0 to V86 mode transfers

All paging operations work correctly. Tests that the processor's paging tables have been set up correctly, and then verifies that the processor correctly translates virtual addresses to physical addresses. Also verifies that the processor correctly manages the page table control flags.

All segmentation operations work correctly. Verifies the correct operation of the segmentation functions. Executed under multiple privilege levels, they identify all segment types, perform base and limit checking, check access control bits, check privilege levels, test all descriptor types in all user modes, and test interrupt gates and descriptors.

The correct interrupt vector is generated for each interrupt. Sets up a handler for each of the 256 interrupt vectors, generates each interrupt via software, and checks that the correct vector handler was invoked.

6.5.2 DEC Alpha Hardware Diagnostics

Processor security functionality tests are available from Microsoft to test the functions of the Alpha 21064-AA CPU that are relevant to the security of Windows NT. These are intended to be periodic diagnostic tests which can be run by the system administrator to verify the proper operation of the processor's protection mechanisms. The Alpha CPU tests are not as extensive as those for the x86 because the Alpha is a much simpler processor. Much of the functionality tested in the x86 is implemented in software on Alpha systems. The following processor features are tested.

TB access controls work correctly. Verify that access control works correctly in all translation buffers (TBs) including at least the kernel- and user-execute permission bits in each instruction TB PTE and at least the kernel and user read and write bits in each data TB PTE.

Page table dirty bit works correctly. The dirty bit functionality must be tested in each DTB PTE array entry.

ASM bit is not stuck on. Verify that the ASM bit works correctly in each of the ITB and DTB PTE array entries. Ensure that the ASM bit is not stuck on.

Virtual to physical page translations are made correctly. Verify that virtual-to-physical page translation works correctly, making sure that each entry in the ITB and DTB PTE arrays are tested.

Super page protection mechanism is correct. Verify that super page translation can only be used in kernel mode.

Correct PALmode vectors are generated. Verify that correct PALmode vectors are generated for at least INTERRUPT, all D-stream errors, ITB_MISS, ITB_ACV, all CALL_PAL (privileged and unprivileged), and OPCDEC entry points.

State changes occur correctly. Verify that the appropriate state changes occur correctly when vectoring to and returning from PALmode for each of the exceptions and interrupts tested above.

Invalid CALL_PAL exceptions are correctly generated. Verify that privileged CALL_PAL exceptions that are executed in a mode other than kernel trap to OPCDEC.

Privileged PALmode instructions are protected. Verify that privileged PALmode instructions (HW_LD, HW_ST, HW_MFPR, HW_MTPR, and HW_REI) cannot be executed in non-PAL mode (either in kernel or user modes).

6.5.3 Compaq Proliant 2000 and 4000 Hardware Diagnostics

The Power On Self Test (POST) resides in ROM and is executed every time the system is powered on. It tests the following system components:

On-board ROM: Verifies checksums on the System and Option ROMs.

Keyboard/Mouse: Checks keyboard and mouse hardware and firmware, stuck keys, cables, and on-board controllers.

System board: Checks DMA, timers, serial and parallel ports, and hardware configuration CMOS RAM checksum.

On-board and expansion memory: Checks amount of RAM present, RAM speed, cache memory and controller, and memory configuration.

Controllers: Checks diskette, SCSI, and video controllers.

Diskette drives: Cycles diskette, verifies drive type.

Hard drives: Checks port assignments, configuration, format, circuitry, cables, drive, drive type, and BIOS data area. Reports previous recoverable and hard errors from history log.

Processor modules: Checks that all processors are present and operating and that their version numbers are correct.

EISA: Checks that each slot is responding, is configured correctly, has the proper card type (EISA or ISA), and initializes correctly.

Each Compaq system is also supplied with the DIAGS system diagnostic program, which can be run at any time by the system administrator to further test peripherals. DIAGS tests the following components and functions:

Processor module(s): CPU functions, interrupt master controller, DMA page registers, CMOS RAM, CMOS clock load data, timer, RAM refresh, protected mode, and clock speed.

Memory: Configuration, ROM checksum, RAM read/write, addressing, walking read/write, and random pattern test.

Keyboard: 8048 (keyboard processor) self-test, typematic test.

Parallel printer: Printer data register, pattern test, connection.

Video display: Controller, video memory, attribute registers, character set, character-mode cell, video modes, and noise.

Diskette drive: Type, format, read, write/read/compare, random seek, media, speed, write protect, change line, pin 34 connected, and media ID.

Serial ports: Port test and clock register.

Fixed drive: Drive type, format, read, read/write/compare, random seek, controller, bad track, head select, conditional format, ECC, drive power, and monitoring.

CD-ROM drive: ID, power, read, media, controller, random seek.

Tape drive: ID, servo write, format, drive sensor, BOT/EOT, read and read/write/compare.

Mouse: interface test.

RAPID is a failure recovery system that attempts to restore the system to operation after a failure without human interaction. The security utility of RAPID is its logging of both fatal and recoverable hardware errors for later analysis. The Server Health Log, stored in non-volatile RAM, consists of the Critical Error Log and the Revision History Table. The Critical Error Log records non-correctable memory errors and catastrophic hardware and software errors that probably caused the system to fail. The Revision History Table allow perusal of the change history of the system configuration stored in battery-backed CMOS RAM. The ART utility can be used to review the Health logs.

Copies of the contents of the last three versions of the EISA configuration files are kept available for review. The ART utility can be used to review the EISA Configuration History files.

6.5.4 DECpc AXP/150 Hardware Diagnostics

The POST resides in ROM and is executed every time the machine is powered on. It tests the following system components:

- System main memory installed on the system board,
- Time-of-year, non-volatile RAM, and NVR battery,
- Serial line controller,
- Interval timer,
- Keyboard controller,
- Video controller, and
- SCSI adapter.

The system administrator may re-run any test explicitly.

Microsoft Hardware Compatibility Tests (HCTs) are available to test the DECpc AXP/150 peripherals. They test and validate the correct operation of the following components:

Hard Disk: Starts multiple synchronous threads that execute reads and writes to and from various sized blocks at random addresses,

Keyboard: Tests a representative set of the keys based on the scan codes returned from key presses.

Mouse: Verifies mouse positioning and single and double clicking on each button. Also verifies that the driver can see the correct number of buttons.

Video (Win GDI): Runs 148 graphics display scripts through a software rasterizer and through the video graphics rasterizer and then verifies that both renderings created the same display memory image.

Async CRC Test: Computes and saves checksum values for a set of existing disk files, then launches several processes which do asynchronous reads and writes to and from these files and re-computes their checksums in the process. The checksums are then verified against those saved in the first pass.

CD-ROM (Copy Compare): Copies files from the CD and then compares them to the original.

Tape I/O: Tests I/O functionality, SetTapePosition and GetTapePosition for Block Moves, and Write Tape Mark.

6.6 Rating Maintenance

This subsection describes the procedures which are in place for the rating maintenance of Windows NT. Individual subsections will describe the Microsoft RAMP process and will specifically identify the personnel involved in the RAMP process, as well as describe how the issues of configuration identification and management, security analysis, tools, RAMP audits, procedures for handling failures of the RAMP process and for handling emergency bug fixes and, finally, some information concerning future RAMP activities.

6.6.1 Personnel Involved in the Microsoft RAMP Process

The personnel that will be involved in the Microsoft RAMP process include the Responsible Corporate Officer (RCO), a Lead Vendor Security Analyst (VSA), two alternate VSAs, a collection of identified Technical Experts (from the development, test, and build groups), and representatives from any involved hardware vendors.

The Lead VSA will assume responsibility for ensuring that all RAMP activities are carried out in accordance with the RM Plan. This will be accomplished through interaction with the alternate VSAs and Technical Experts. RAMP audits will also contribute to ensuring that all activities are performed, these audits will be performed by one of the alternate VSAs.

6.6.2 Scope of Product Changes to be Addressed via RAMP

The Microsoft RAMP process is designed to address changes incorporated into Windows NT via Service Packs,[2] to add new hardware platforms within the current micro-processor families, and to add new micro-processor families to the evaluated configuration of Windows NT.

It is not intended that major releases of Windows NT will undergo a RAMP process. The scope of change is too large to be feasibly accommodated by the defined RAMP activities.

6.6.3 RAMP Process Overview

All Windows NT product evidence necessary to support the awarded rating (i.e., documentation, source code, test evidence) is covered by the procedures defined in the Windows NT Rating Maintenance Plan. This subsection addresses each of these areas. Peer level code review is used to approve software changes and database management tools are used to track software changes.

6.6.3.1 Windows NT Documentation

The procedures for maintaining the user and design documentation for Windows NT differ. For user documentation (i.e., SFUG and TFM), the Windows NT Documentation group will make any necessary changes

[2] Service Packs are Microsoft's mechanism for providing minor updates between major releases of Windows NT.

based on the changes to the Security-Relevant[3] product changes. These documents will then go through iterations of review by the VSAs and technical experts from the Windows NT Development group.

For design documentation, the documentation is categorized similarly to the code to which it pertains (i.e., based on security relevance). Changes to all Security-Relevant documentation will be listed and summarized in the Rating Maintenance Report. Changes to Non-Security-Relevant documentation will be reviewed by the VSA but will not be reflected in the RMR.

6.6.3.2 Test Evidence

Testing responsibility lies in the test group. The test group tracks all changes to Windows NT to ensure that existing tests are updated and new tests are created when necessary. Specifically, new tests will be created whenever new functionality is added to Windows NT. Existing tests will be reviewed to determine if revision is necessary whenever existing functionality is modified. Tests will be updated as necessary to address bug fixes.

The VSA will review all test evidence to ensure that sufficient test coverage is maintained for the Windows NT TCB.

6.6.3.3 Evidence Consistency

The consistency of product evidence will be maintained through the use of the documentation of the security analysis. A template exists which defines the areas that must be covered during the security analysis. This template includes a description of the change, as well as the implications for related documentation and test evidence. The security evaluation of each change will ensure that all product evidence (i.e., code, documentation, tests) is kept consistent with one another.

6.6.4 Configuration Identification and Control

The granularity of identification and control of the Windows NT TCB software is the individual source code file. Each proposed change to a source file is identified as either:

- *Non-Security-Relevant:* meaning that the file should not have any affect on the security of Windows NT.
- *Security Relevant:* meaning that the source file affects the security of Windows NT.

For each subsystem of Windows NT, the non-security-relevant changes will be treated as a single configuration item. The security relevant changes will each be maintained as a separate configuration item.

The level of security analysis then performed on changes to the files will be scoped according to the security relevance of the changes. Specifically:

[3] Definition provided in Section "Configuration Identification and Control," on page 175

- For *non-security-relevant changes,* individual changes will be accompanied by a statement as to why the change is not security relevant. A cumulative analysis, encompassing all changes, will then be performed to ensure that the non-security-relevant modules need not be reclassified to either of the security relevant categories.
- For *all security-relevant changes,* an explanation of why the change does not impact the rating of Windows NT will be given. A cumulative analysis will also be performed to ensure that the interaction of the changes does not cause a degradation of the Windows NT rating.

6.6.5 Security Analysis

Microsoft will provide training on the RAMP requirements for security analysis to the Windows NT development staff who will serve as the technical experts supporting RAMP. Individual members of the development staff will then be responsible for ensuring that security analysis is performed for all changes made to the system. VSAs will be used to oversee and audit the security analysis performed by the development staff.

6.6.6 Adherence to Interpretations

Formal interpretations of the TCSEC will be reviewed by the VSAs and other technical experts within the Windows NT development group on a periodic (e.g., monthly) basis. A decision will be made for each interpretation whether there is any affect on Windows NT that would affect the assigned rating. If so, a course of action to be taken to address the interpretation will be determined.

The Rating Maintenance Report will describe, for each interpretation published since the award of the rating or the last RAMP cycle, a justification as to why the interpretation does or does not affect Windows NT. If the interpretation is determined to affect Windows NT, a description of the changes made to address the interpretation will be provided.

6.6.7 RAMP Audits

RAMP audits will begin with the use of the WinDiff tool to identify all changes made between two versions of Windows NT. This tool will identify files that have been added or deleted since the previous version as well as the specific changes that have been made to other files.

Using the output of this tool, the VSA will review all changes made to security-relevant files and their associated documentation. A sampling will be done of the changes made to the non-security-relevant files. In addition, particular attention will be paid to the review of deleted and new source files to ensure that they are appropriately labeled with respect to their security relevance.

6.6.8 RAMP Process Failures

The following steps will be taken if a failure of the RAMP process for Windows NT is identified:

- The cause of the process failure will be determined.

- The process flaw will be corrected and the RM Plan updated.
- All security analysis that could have been negatively affected by the process failure will be re-done.

6.6.9 Hardware Ports

Microsoft expects that the most significant RAMP activity will center around the inclusion of additional hardware platforms into the evaluated configuration of Windows NT. Personnel from the hardware manufacturer will serve as technical experts to perform any necessary security analysis and will be used to prepare the necessary documentation to support the RAMP activity. Microsoft envisions many additional hardware platforms to be added through this manner.

6.6.10 Emergency Bug Resolution

A quick fix team will be used to correct high priority problems. The quick fix team will consist of a member from the development and test groups that will be assigned to particular areas of Windows NT. When a problem is identified, it will be entered into a problem report database so that it can be tracked and audited. The quick fix team will reproduce and correct the problem and test the correction. The fix will then be distributed by the Product Support Services group. Depending on the severity of the problem, the fix, or a notice of its availability, may also be posted to Compuserve.

All emergency bugs handled by the quick fix team will be subject to the same security analysis and ramp audit processes used for other bug fixes or feature enhancements.

This page intentionally left blank

Chapter 7
Evaluation as a C2 System

7.1 Discretionary Access Control

Requirement

The TCB shall define and control access between named users and named objects (e.g., files and programs) in the ADP system. The enforcement mechanism (e.g., self/group/public controls, access control lists) shall allow users to specify and control sharing of those objects by named individuals, or defined groups of individuals, or by both, and shall provide controls to limit propagation of access rights. The discretionary access control mechanism shall, either by explicit user action or by default, provide that objects are protected from unauthorized access. These access controls shall be capable of including or excluding access to the granularity of a single user. Access permission to an object by users not already possessing access permission shall only be assigned by authorized users.

Applicable Features

Windows NT uses DACLs to control access between named users and named objects. The use of DACLs allows users to specify and control the sharing of objects with other users, groups of users, or both. DACLs and DACL evaluation is discussed further in Section "Access Control List Data Structure," on page 72 and in Section "Access Validation Algorithms," on page 73. The DACL structure has the ability to specify that an access right be granted or denied at the granularity of an individual user. All objects and their allowed access rights are described in Section "Overview of Object Access Rights," on page 142.

There are rules which dictate how an object is assigned a DACL when it is created. These rules are described in Section "ACL Inheritance," on page 146. If no DACL is provided through various other ways, the object will always receive the default DACL which is a field in the object's owner's token. Therefore, objects are protected by default. An object's default protection is further described in Section "Default Object Security," on page 148.

Access permission to an object by users not already possessing access permission can only be granted by modifying the DACL of that object to grant this access permission. The owner of an object always has ReadControl and WriteDAC access which allow the owner to always be able to read and write the DACL. The owner of an object and those with WriteDAC access to an object are the only users who can modify the DACL of that object. Those with WriteOwner access have the ability to modify the DACL, but they must first make themselves the owner of the object. Object Ownership is described in further detail in Section "Object Ownership," on page 145. Propagation of access rights is limited by object ownership.

There are several privileges which allow exception to the DACL: TakeOwnership, Backup, Restore, and Debug. These are each described in Section "DAC Related Privileges," on page 146. Of these privileges,

Debug will not be assigned to any users in the evaluated configuration and the TFM has instructions that the others should only be assigned to administrators.

Conclusion

Windows NT satisfies the C2 Discretionary Access Control requirement.

7.2 Object Reuse

Requirement

All authorizations to the information contained within a storage object shall be revoked prior to initial assignment, allocation or reallocation to a subject from the TCB's pool of unused storage objects. No information, including encrypted representations of information, produced by a prior subject's actions is to be available to any subject that obtains access to an object that has been released back to the system.

Applicable Features

All resources, including all Executive and Non-Executive objects, visible at the TCB interface are appropriately handled with respect to object reuse.

Executive objects are handled by either or both, the Object Manager and the specific Executive subsystem which is responsible for maintaining and manipulating the object type (e.g., the Process Manager for processes and threads).

Non-Executive Objects are handled by the non-executive trusted server which exports the object to untrusted clients. Object reuse protection is attained through either the clearing of objects when they are first created or by the overwriting of new information over old at creation time. Additionally, pages of memory and secondary storage are handled by the Memory Manager and I/O Manager respectively. For specific details about individual objects, refer to Section "Object Reuse," on page 153.

Conclusion

Windows NT satisfies the C2 Object Reuse requirement.

7.3 Identification and Authentication

Requirement

The TCB shall require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate. Furthermore, the TCB shall use a protected mechanism (e.g., passwords)

to authenticate the user's identity. The TCB shall protect authentication data so that it cannot be accessed by any unauthorized user. The TCB shall be able to enforce individual accountability by providing the capability to uniquely identify each individual ADP system user. The TCB shall also provide the capability of associating this identity with all auditable actions taken by that individual.

Applicable Features

In Windows NT, each user must identify him or herself with a username to logon to the system. Each user is uniquely identified by a username which is represented internally by a SID. This SID is unique and can not be reassigned to any one else. Using the SID in the process primary token, Windows NT associates a process to a particular user. The audit records generated by the TCB contain the SID (and hence the username) in all user related security events.

Windows NT allows every user to have a corresponding password with the username. The TFM describes how to assure that all logon accounts require passwords. The passwords are stored in the SAM database (see Section "Security Accounts Manager," on page 112) and are encrypted. Any operation on the SAM database requires access to the objects containing these data which are protected by DAC. The username and the corresponding password are authenticated via the LSA protected server (see Section "Local Security Authority," on page 109) assuring that only authorized users can logon to the system.

Conclusion

Windows NT satisfies the C2 Identification and Authentication requirement.

7.4 Audit

Requirement

The TCB shall be able to create, maintain, and protect from modification or unauthorized access or destruction an audit trail of accesses to the objects it protects. The audit data shall be protected by the TCB so that read access to it is limited to those who are authorized for audit data. The TCB shall be able to record the following types of events: use of identification and authentication mechanisms, introduction of objects into a user's address space (e.g., file open, program initiation), deletion of objects, actions taken by computer operators and system administrators and/or system security officers, and other security relevant events. For each recorded event, the audit record shall identify: date and time of the event, user, type of event, and success or failure of the event. For identification/authentication events the origin of request (e.g., terminal ID) shall be included in the audit record. For events that introduce an object into a user's address space and for object deletion events the audit record shall include the name of the object. The ADP system administrator shall be able to selectively audit the actions of any one or more users based on individual identity.

Applicable Features

Windows NT's TCB creates and maintains an audit trail of access to objects it protects and other security relevant events. The audit mechanism is made of the following components: the LSA, the SRM, protected servers and executive subsystems, Event Logger, and the Event Viewer. The mechanism is described in Section "Audit Mechanism," on page 158. Audit information is recorded in the security log which is protected from modification and unauthorized access by its DACL which grants read and write access to members of the Administrative group only. As stated in Section "DAC Related Privileges," on page 146, the Security privilege also allows access to the security log. This privilege is only given to Administrators. The security log is written to a file which is protected via its DACL which gives read and write access only to members of the Administrators group.

The auditable events are listed in Table 5.7, on page 160. The contents of the header of each audit record is provided in Section "Audit Record Contents," on page 164. Identification and authentication audit records include the computer name which the audit originated from. Object Access audit records include the object name.

The audit mechanism allows for an administrator to pre-select the audit events in the system. The categories of events that can be pre-selected are: system, logon/logoff, object access, privilege use, policy changes, and account management. These are further described in Section "Audit Mechanism," on page 158.

The administrator uses the Event Viewer to view the security log. Access to the security log, which requires the Security privilege or permission via the DACL, is needed to use this tool to view the security log. The Event Viewer is described further in Section "Event Viewer and Audit," on page 161 and in Section "Event Viewer," on page 129. This tool allows the administrator to selectively view the audit actions of users based on individual identity.

The security log can be configured to prevent the loss of audit records. This is described in Section "Audit Loss Prevention," on page 164.

Note that Windows NT satisfies the object deletion requirement as interpreted by the Audit Guideline and not as according to the strict interpretation of the TCSEC.

Conclusion

Windows NT satisfies the C2 Audit requirement.

7.5 System Architecture

Requirement

The TCB shall maintain a domain for its own execution that protects it from external interference or tampering (e.g., by modification of its code or data structures). Resources controlled by the TCB may be a defined subset of the subjects and objects in the ADP system. The TCB shall isolate the resources to be protected so that they are subject to the access control and auditing requirements.

Applicable Features

The software portion of the TCB uses several execution domains to protect itself from external interference and tampering. The Executive runs in the privileged model (i.e., kernel-mode) provided by both evaluated processor architectures. All TCB protected servers and Administrator tools execute in a TCB process, which is protected through the address space isolation mechanisms of the Executive. On-disk storage for TCB code and data structures are all protected via the DACL mechanism provided by the Executive.

The TCB protects all system resources, and subjects all attempts to access these resources to both the access control and audit mechanisms.

Conclusion

Windows NT satisfies the C2 System Architecture requirement.

7.6 System Integrity

Requirement

Hardware and/or software features shall be provided that can be used to periodically validate the correct operation of the on-site hardware and firmware elements of the TCB.

Applicable Features

The Compaq Proliant 2000 and 4000 performs system POST at start-up, and comes packaged with a system and peripherals test suite called DIAGS and a hardware error logging and reporting system. Microsoft can also supply, on request, processor diagnostic tests for the Pentium processor. Descriptions of the Compaq Proliant 2000 and 4000 system diagnostic tests and the Intel Pentium CPU diagnostic tests can be found in Section "Compaq Proliant 2000 and 4000 Hardware Diagnostics," on page 171.

The DEC AXP/150 has a POST that automatically executes at startup and can also be run by the administrator on demand. Microsoft can provide its hardware compatibility test suite for more extensive testing of the system and peripherals, and a set of processor diagnostic tests for testing the functionality of the CPU. Descriptions of the DECpc AXP/150 system diagnostic tests and the DEC Alpha CPU diagnostic tests can be found in Section "DECpc AXP/150 Hardware Diagnostics," on page 173.

Conclusion

Windows NT satisfies the C2 System Integrity requirement.

7.7 Security Testing

Requirement

The security mechanisms of the ADP system shall be tested and found to work as claimed in the system documentation. Testing shall be done to assure that there are no obvious ways for an unauthorized user to bypass or otherwise defeat the security protection mechanisms of the TCB. Testing shall also include a search for obvious flaws that would allow violation of resource isolation, or that would permit unauthorized access to the audit or authentication data.

Applicable Features

The vendor's testing philosophy is described in Section "Security Testing," on page 169. The vendor's test suite was executed on both the Windows NT Server and Workstation products on each hardware platform in the evaluated configuration. The team analyzed the test results and the tests were reran until all anomalies were resolved. In addition to the vendor tests, the team developed and executed tests and performed additional activities for increased confidence. The Hardware Integrity Tests were run to satisfaction on each hardware platform in the evaluated configuration.

Conclusion

Windows NT satisfies the C2 Security Testing requirement.

7.8 Security Features User's Guide

Requirement

A single summary, chapter, or manual in user documentation shall describe the protection mechanisms provided by the TCB, guidelines on their use, and how they interact with one another.

Applicable Features

The Windows NT Security Features Users Guide (SFUG) describes the protection mechanisms provided by the TCB. It describes how to logon to Windows NT, how to change a user's password, how to lock and unlock the computer, and how to use a screen saver. The SFUG also describes how to place DACLs on files and directories and explains how ownership of files and directories is achieved and what ownership means (see Section "Object Ownership," on page 145).

Conclusion

Windows NT satisfies the C2 Security Features User's Guide requirement.

7.9 Trusted Facility Manual

Requirement

A manual addressed to the ADP system administrator shall present cautions about functions and privileges that should be controlled when running a secure facility. The procedures for examining and maintaining the audit files as well as the detailed audit record structure for each type of audit event shall be given.

Applicable Features

The Windows NT TFM presents cautions about functionality and informs the administrator of the privileged built-in groups whose membership should be controlled. The privileges belonging to each built-in group is described and the special abilities they provide is also discussed. The procedures for examining and maintaining the audit file (security log) is provided along with the record structure for each type of audit event. The TFM also describes the purpose of the administrative tools.

Conclusion

Windows NT satisfies the C2 Trusted Facility Manual requirement.

7.10 Test Documentation

Requirement

The system developer shall provide to the evaluators a document that describes the test plan, test procedures that show how the security mechanisms were tested, and results of the security mechanisms' functional testing.

Applicable Features

The vendor's Test Documentation is described in Section "Test Documentation," on page 169.

The test documentation was found to be adequate and sufficient with respect to both breadth and depth of coverage.

Conclusion

Windows NT satisfies the C2 Test Documentation requirement.

7.11 Design Documentation

Requirement

Documentation shall be available that provides a description of the manufacturer's philosophy of protection and an explanation of how this philosophy is translated into the TCB. If the TCB is composed of distinct modules, the interfaces between these modules shall be described.

Applicable Features

Windows NT, including its security features and mechanisms, is a well documented system. The philosophy of protection is described in "Windows NT - Philosophy of Protection," among other documents. The realization of this philosophy of protection in the TCB is described in Inside Windows NT by Helen Custer. The interfaces between the different modules of the TCB is detailed in dozens of design specification documents. These are listed in "Bibliography and References," on page 195 under the heading, "Windows NT Design Documentation References." Although it is not required for C2 certification, the security model used by Windows NT is described in the Windows NT Resource Kit.

Conclusion

Windows NT satisfies the C2 Design Documentation requirement.

7.12 Additional Security Features

7.12.1 B2 Trusted Facility Management

Requirement

The TCB shall support separate operator and administrator functions.

Applicable Features

Windows NT provides the ability to grant an arbitrary subset of rights to a user, allowing a given site the ability to define the role of "operator" according to local needs.

In addition, built-in groups, like Backup Operator and Power Users allow users to perform well-defined roles without providing the full range of capabilities associated with administrator accounts. For example, although the backup operator's privilege permits him to override the ACLs on files, it does so only in the context of the Windows NT-provided backup tool. A person possessing the backup privilege may not peruse arbitrary files. Similarly, members of the Power Users group can create groups, and add user accounts.

Conclusion

Although Windows NT satisfies this functional requirement at the B2 level, Windows NT was not evaluated against any assurance requirements above its rated level.

Windows NT satisfies the B2 Trusted Facility Management requirement.

7.12.2 B2 Trusted Path

Requirement

The TCB shall support a trusted communication path between itself and users for initial login and authentication. Communications via this path shall be initiated exclusively by a user.

Applicable Features

Windows NT provides a trusted path for logon and authentication. It is initiated by the Secure Attention Sequence (pressing the Ctrl, Alt, and Delete keys simultaneously). Executing the SAS guarantees that communication will be with the WinLogon server, not any other process, in order to logon, logoff, shutdown, lock the workstation, or change password. Non-privileged users can only change their passwords through the trusted path.

Conclusion

Although Windows NT satisfies this functional requirement at the B2 level, Windows NT was not evaluated against any assurance requirements above its rated level.

Windows NT satisfies the B2 Trusted Path requirement.

This page intentionally left blank

Chapter 8

Evaluator's Comments

8.1 Robustness of Security When Part of Initial Design

One of the major initial design goals for Windows NT was to assure C2-level security through an integral, uniform protection mechanism. All system resources are treated as objects, and thus a single security "gate" can be the protection component that all users must pass through to acquire system resources. This results in much greater assurance that the system meets the applicable security criteria, because a single security mechanism is easier to understand and to verify than multiple ad hoc mechanisms. When security is not an absolute requirement of the initial design, it is virtually impossible through later add-ons to provide the kind of uniform treatment to diverse system resources that Windows NT provides.

8.2 Object Deletion Auditing

The audit record indicating opening an object also list the particular access right(s) granted to the user for that object. Each particular access right grants the user the ability to perform a defined set of actions. There is a delete access right which allows a user to delete an object. Even though the audit record may list this access right in the open audit record, this does not necessarily mean that the user actually deleted the object.

This page intentionally left blank

Appendix A
Evaluated Hardware Components

Windows NT is designed to run on a variety of machines of various architectures. One of the evaluated machines (Compaq) uses the International Business Machines, Inc. (IBM) PC/AT compatible architecture. The other (DEC) uses an architecture similar to the PC/AT, but with a DEC Alpha processor instead of an $x86$ processor.

A.1 Compaq Proliant 2000 and 4000

The Compaq models are the Proliant 2000 and the Proliant 4000 5/90-1 and 5/100-1, all of which are designated as file servers, but are evaluated for use as stand-alone workstations only, with no networking capability (regardless of whether the Windows NT Workstation or Server system is installed). They can be configured with from one to four processor modules.

Component	Description	Compaq Part Number
System board	Tri-Flex EISA	Tri-Flex
Processor Module	Pentium, 90 MHz	199050-001
	Pentium, 100 MHz	163888-001
Memory	16 MB, 70ns kit	149949-001
	32 MB, 70ns kit	149912-001
	64 MB, 70ns kit	149913-001
	128 MB, 70ns kit	149914-001
Video Adapter	SVGA 1024x768x4	(Onboard)
SCSI Controller	32-bit FAST SCSI-2	142013-001
Fixed Media Drive	1.5 GB SCSI-2	146742-003
	2.1 GB SCSI-2	146742-004
Removable Media	CD-ROM	142193-001
	3.5" 1.44 MB FDD	113638-001
	2/8 GB DAT Drive	142019-001
Keyboard	101 Key English	160650-101
Mouse	3-button	141649-002
Printer	HP LaserJet IV	

A.2 DECpc AXP/150

The DEC system is the DECpc AXP/150, which is evaluated for use as a stand-alone workstation, without networking capability (regardless of whether the Windows NT Workstation or Server system is installed). The ROM Console Firmware on the system board must have a version number of 3.5-5 or greater.

Component	Description	Digital Part Number
System board	21064-AA CPU, 150 MHz	54-20674-04
Memory	16 MB SIMMs	PB2MA-BD
	32 MB SIMMs	PC77M-AB
	64 MB SIMMs	PB2MA-BF
	128 MB SIMMs	PC77M-AD
Video Adapter	Compaq Qvision SVGA	PB2GA-AA
SCSI Controller	Adaptec 1742	PB2HA-SA
Fixed Media Drive	535 MB SCSI	PB2RA-EA
Removable Media	CD-ROM RRD42	PB2SA-AA
	3.5" 1.44 MB FDD	RX26-AA
Keyboard	101 Key English	PCXAL-FA
Mouse	3-button	PCXAS-AA
Printer	HP LaserJet IV	

Appendix B
Evaluated Software Components

The evaluated software product is the Microsoft Windows NT Workstation and Server Version 3.5. To meet the C2 requirements, the administrator shall follow the TFM guidance to disable the OS/2 and POSIX subsystems. Also, the evaluated configuration excludes Windows NT's networking capabilities.

Service Pack 3, which addresses bug fixes to Version 3.5, must also be installed.

This page intentionally left blank

Appendix C
Bibliography and References

[1] *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 520028-STD, December 1985

Evaluation Documentation References: Hardware Documents

[2] *DECchip 21064-AA Microprocessor, Hardware Reference Manual*, Digital Equipment Corporation, October 1992.

[3] *Alpha Architecture Reference Manual*, Digital Equipment Corporation, 1992.

[4] *DECchip 21064-150, -166, -200 Alpha AXP Microprocessor Data Sheet (EB-N0136-72)*, Digital Equipment Corporation, 1993.

[5] *DECchip 21064A-233, -275 Alpha AXP Microprocessor Data Sheet (EC-N0699-72)*, Digital Equipment Corporation, 1993.

[6] *Answers to Common Questions about PALcode for Alpha AXP Systems (EC-N0647-72)*, Digital Equipment Corporation, 1993.

[7] *Alpha Architecture Handbook (no order no.)*, Digital Equipment Corporation, 1993.

[8] *Pentium Processor User's Manual, Volume 3, Architecture and Programming Manual*, Intel Corporation, 1993.

[9] *i486 Processor User's Manual, Volume 3, Architecture and Programming Manual*, Intel Corporation, 1993.

[10] *Peripheral Components*, Intel Corporation, 1994.

[11] *82495XP Cache Controller & 82490XP Cache RAM Data Book*, Intel Corporation, June 1991.

[12] *Jensen, EISA Bus Alpha PC, Hardware and Software Programming Guide*, Digital Equipment Corporation, April 1992.

[13] *PB22H-KB System Module Hardware Reference Manual*, Digital Equipment Corporation, July, 1993.

User Documentation

[14] *System Guide - Microsoft Windows NT* Microsoft Corporation, April 1993.

[15] *Microsoft Win32 Programmer's Reference, Volumes I-IV*, Microsoft Corporation, 1993.

[16] *Microsoft Windows NT - Enterprise Planning Guide: Security*, Microsoft Corporation, 1994.

Appendix D

List of Acronyms

ACE	Access Control Entry
ACL	Access Control List
AECC	Advanced Error Correcting Code
AMC	Aruba Memory Controller
APC	Asynchronous Procedure Call
ASIC	Application Specific Integrated Circuit
ASN	Address Space Number
BCD	Binary Coded Decimal
BIOS	Basic I/O Supervisor
BIU	Bus Interface Unit
CDFS	CD-ROM File System
CD-ROM	Compact Disk Read Only Memory
CPL	Current Privilege Level
CPU	Central Processing Unit
CSP	Central System Peripheral
DAC	Discretionary Access Control
DACL	Discretionary Access Control List
DD	Device Drivers
DDE	Dynamic Data Exchange
DLL	Dynamic Link Libraries
DMA	Direct Memory Access
MS-DOS	Microsoft Disk Operating System
DPC	Deferred Procedure Call
DPL	Descriptor Priority Level
DRAM	Dynamic Random Access Memory
DSP	Distributed System Peripheral
DTB	Data Translation Buffer
EBB	EISA Bus Buffer
EBC	EISA Bus Controller
EEPROM	Electrically Erasable Programmable Read Only Memory
EISA	Extended Industry Standard Architecture
FAT	File Allocation Table
FCB	File Control Block
FEPROM	Flash Erasable Programmable Read Only Memory
FSD	File System Drivers
GB	Gigabyte
GDI	Graphics Display Interface
GDT	Global Descriptor Table
GDTR	Global Descriptor Table Register
HAE	Host Address Extension

HAL	Hardware Abstraction Layer
HCT	Hardware Compatibility Test
IDT	Interrupt Dispatch Table
IDTR	Interrupt Descriptor Table Register
IOPL	I/O Privilege Level
IPC	Interprocess Communication
IRP	I/O Request Packets
IRQL	Interrupt Request Level
ISA	Industry Standard Architecture
ISP	Integrated System Peripheral
ISR	Interrupt Service Routine
ITB	Instruction Translation Buffer
KB	Kilobyte
LCN	Logical Cluster Number
LDT	Local Descriptor Table
LDTR	Local Descriptor Table Register
LPC	Local Procedure Call
LSA	Local Security Authority
MB	Megabyte
MESI	Modified/Exclusive/Shared/Invalid
MFT	Master File Table
MHz	Megahertz
MSFS	Mailslot File System
NMI	Non-Maskable Interrupt
NPFS	Named Pipe File System
NSA	National Security Agency
NTFS	Windows NT File System
NVRAM	Non-Volatile Random Access Memory
OS/2	Operating System 2
PC	Personal Computer
PC/AT	Personal Computer/Advanced Technology
PDE	Page Directory Entry
PFN	Page Frame Number
PID	process ID
POSIX	Portable Operating System Interface
POST	Power On Self Test
PRCB	Processor Control Block
PTE	Page Table Entry
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RPC	Remote Procedure Call
RPL	Requested Priority Level
SACL	System Access Control List
SAM	Security Accounts Manager
SAS	Secure Attention Sequence
SCB	Stream Control Block
SCSI	Small Computer Standard Interface

SDB	Super Data Buffer
SID	Security Identifier
SIMM	Single Inline Memory Module
SQOS	Security Quality of Service
SRM	Security Reference Monitor
SROM	Serial Read Only Memory
SYSCTL	System Control Register
TB	Translation Buffer
TCB	Trusted Computer Base
TCSEC	Trusted Computer System Evaluation Criteria
TFM	Trusted Facilities Manual
TLB	Translation Lookaside Buffer
TSS	Task State Segment
VAD	Virtual Address Descriptor
VCN	Virtual Cluster Number
VDM	Virtual DOS Machine
VMM	Virtual Memory Manager
VPB	Volume Parameter Block